

OS/390



C/C++ SOM-Enabled Class Library User's Guide and Reference

OS/390



C/C++ SOM-Enabled Class Library User's Guide and Reference

Note !

Before using this information and the product it supports, be sure to read the general information under "Notices" on page ix.

Third Edition, September 1997

This edition applies to Version 2 Release 4 of IBM OS/390 C/C++ (Program 5647-A01) and to all subsequent releases and modifications until otherwise indicated in new editions or other updated documentation. Make sure that you are using the correct edition for the level of the product. Also, make sure that you apply all necessary PTFs for the level of the product.

Changes or additions to the text and illustrations are indicated by a vertical line (|) to the left of the change or addition.

Order publications through your IBM representative or the IBM branch office serving your locality. Publications are not stocked at the address given below.

IBM welcomes your comments. A form for readers' comments is provided at the back of this publication. If the form has been removed, address your comments to:

IBM Canada Ltd. Laboratory
Information Development
2G/345/1150/TOR
1150 Eglinton Avenue East
North York, Ontario, Canada. M3C 1H7

You can also send your comments by facsimile (attention: RCF Coordinator) or you can send your comments electronically to IBM. See "Communicating Your Comments to IBM" for a description of the methods. This information immediately precedes the Readers' Comment Form at the back of this publication.

If you send comments, include the title and order number of this book, and the page number or topic related to your comment.

When you send information to IBM, you grant IBM a nonexclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

© Copyright International Business Machines Corporation 1995, 1997. All rights reserved.

Note to U.S. Government Users — Documentation related to restricted rights — Use, duplication or disclosure is subject to restrictions set forth in GSA ADP Schedule Contract with IBM Corp.

Contents

Notices	ix
Programming Interface Information	ix
Standards	ix
Trademarks	x
 About this Book	 xiii
IBM OS/390 C/C++ and Related Publications	xiii
Hardcopy Books	xix
Softcopy Books	xix
Softcopy Examples	xx
For Late Breaking C/C++ News...	xxi
OS/390 C/C++ on the World Wide Web	xxi
 About IBM OS/390 C/C++	 xxiii
The C/C++ Compilers	xxiii
The C Language	xxiii
The C++ Language	xxiii
Common Features of the OS/390 C and C++ Compilers	xxiv
Features Specific to the OS/390 C Compiler	xxv
Features Specific to the OS/390 C++ Compiler	xxv
Utilities	xxvi
Class Libraries	xxvi
Class Library Source	xxvii
The Debug Tool	xxvii
OS/390 Language Environment	xxviii
OS/390 OpenEdition	xxix
OS/390 OpenEdition Services	xxix
OS/390 C/C++ Applications with OpenEdition C/C++ Functions	xxx
Input and Output	xxxi
I/O Interfaces	xxxi
File Types	xxxii
Additional I/O Features	xxxiii
The System Programming C Facility	xxxiv
Interaction with Other IBM Products	xxxiv
Additional Features of OS/390 C/C++	xxxv
 C++SOM and Cross-language SOM Class Libraries	 3
 Chapter 1. C++ SOM and Cross-language SOM Class Libraries	 5
SOM-enabled and Not SOM-enabled Versions	5
Why Multiple Library Versions?	6
C++ SOM and Cross-language SOM Collection Classes	6
Coding with Class Libraries under OS/390 OpenEdition Services	7
Compiling and Binding with the C++ SOM Libraries	7
Migration Notes	9
Using the Cross-language SOM Collection Class Library	9
 User's Guide: SOM Cross-language Collection Classes	 11

Chapter 2. Overview of the SOM Cross-language Collection Classes	13
Classes Provided by the Library	13
Benefits of the SOM Cross-language Collection Classes	17
Types of Classes in the SOM Cross-language Collection Classes	17
Flat Collections	18
Ordering of Collection Elements	19
Access by Key	19
Equality for Keys and Elements	20
Uniqueness of Entries	22
Restricted Access	24
Auxiliary Classes	24
The Overall Implementation Structure	24
Abstract Classes	25
Chapter 3. Using the Collection Classes	27
Creating an Operations Class Object	27
Creating Collections	28
Adding, Removing, and Replacing Elements	28
Adding Elements	28
Removing Elements	29
Replacing Elements	29
Cursors	30
Using Cursors for Locating and Accessing Elements	31
Iterating over Collections	31
Iteration Using Cursors	32
Iteration Using Applicators	32
Bounded and Unbounded Collections	33
Chapter 4. Element Functions and Key-Type Functions	35
Introduction to Element Functions and Key-Type Functions	35
Chapter 5. Polymorphic Use of Collections	37
Introduction to Polymorphism	37
Chapter 6. Exception Handling	39
Introduction to Exception Handling	39
Exceptions Caused by Violated Preconditions	39
Exceptions Caused by System Failures and Restrictions	40
Levels of Exception Checking	40
List of Exceptions	40

Reference: SOM Cross-language Collection Classes - Flat Collections 43

Chapter 7. Introduction to Flat Collections	45
Terms Used	45
Chapter 8. Flat Collection Member Functions	47
Chapter 9. Bag	73
Chapter 10. Deque	75
Chapter 11. Equality Sequence	77

Chapter 12. Heap	79
Chapter 13. Key Bag	81
Chapter 14. Key Set	83
Chapter 15. Key Sorted Bag	85
Chapter 16. Key Sorted Set	87
Chapter 17. Map	89
Chapter 18. Priority Queue	91
Chapter 19. Queue	93
Chapter 20. Relation	95
Chapter 21. Sequence	97
Chapter 22. Set	99
Chapter 23. Sorted Bag	101
Chapter 24. Sorted Map	103
Chapter 25. Sorted Relation	105
Chapter 26. Sorted Set	107
Chapter 27. Stack	109

Reference : SOM Cross-language Collection Classes - Auxiliary Classes 111

Chapter 28. Global	113
Chapter 29. Cursor	115
Public Member Functions	115
Chapter 30. Applicator	119
Chapter 31. Comparator	121
Chapter 32. Predicate	123
Chapter 33. Operations	125

Reference : SOM Cross-language Collection Classes - Abstract Classes 127

Chapter 34. Collection	129
Chapter 35. Equality Collection	131

Chapter 36. Key Collection	133
Chapter 37. Ordered Collection	135
Chapter 38. Sorted Collection	137
Chapter 39. Sequential Collection	139
Chapter 40. Equality Key Collection	141
Chapter 41. Key Sorted Collection	143
Chapter 42. Equality Sorted Collection	145
Chapter 43. Equality Key Sorted Collection	147
Appendix A. Coding Samples: Source Code and Header Files	149
Coding Example for Deque	149
Coding Example for Key Bag	156
Coding Example for Set	168
Coding Example for Sorted Set	175
Coding Example for Stack	186
Index	193

Figures

1. Libraries in OS/390 Language Environment	xxviii
2. Combination of Flat Collection Properties	19
3. Behavior of add for Unique and Multiple Collections	23
4. Abstract Hierarchy	26
5. Abstract Hierarchy of Flat Collections with Restricted Access	26

Notices

Any reference to an IBM licensed program in this publication is not intended to state or imply that only IBM's licensed program may be used. Any functionally equivalent product, program, or service that does not infringe any of IBM's intellectual property rights may be used instead of the IBM product, program, or service. Evaluation and verification of operation in conjunction with other products, except those expressly designated by IBM, is the user's responsibility.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to the IBM Director of Licensing, IBM Corporation, 500 Columbus Avenue, Thornwood, NY, 10594, USA.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independent created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact IBM Canada Ltd., Department 071, 1150 Eglinton Avenue East, North York, Ontario M3C 1H7, Canada. Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

This publication contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

Any interfaces, including service component interfaces, that are not documented in the OS/390 C/C++ publications are not formal interfaces. You should not build any dependencies on these interfaces, as IBM can change or remove interfaces at any time, without notice.

Programming Interface Information

This book documents General-Use Programming Interface and associated guidance information provided by the IBM OS/390 C/C++ and OS/390 Language Environment products.

General-Use Programming Interfaces allow the customer to write programs that obtain the services of the IBM OS/390 C/C++ compiler and IBM OS/390 Language Environment.

Standards

Extracts are reprinted from *IEEE Std 1003.1—1990, IEEE Standard Information Technology—Portable Operating System Interface (POSIX)—Part 1: System Application Program Interface (API) [C language]*, copyright 1990 by the Institute of Electrical and Electronic Engineers, Inc.

Extracts are reprinted from *IEEE P1003.1a Draft 6 July 1991, Draft Revision to Information Technology—Portable Operating System Interface (POSIX), Part 1:*

System Application Program Interface (API) [C Language], copyright 1992 by the Institute of Electrical and Electronic Engineers, Inc.

Extracts are reprinted from *IEEE Std 1003.2—1992, IEEE Standard Information Technology—Portable Operating System Interface (POSIX)—Part 2: Shells and Utilities*, copyright 1990 by the Institute of Electrical and Electronic Engineers, Inc.

Extracts are reprinted from *IEEE Std P1003.4a/D6—1992, IEEE Draft Standard Information Technology—Portable Operating System Interface (POSIX)—Part 1: System Application Program Interface (API)—Amendment 2: Threads Extension [C language]*, copyright 1990 by the Institute of Electrical and Electronic Engineers, Inc.

Extracts from *ISO/IEC 9899:1990* have been reproduced with the permission of the International Organization for Standardization (ISO) and the International Electrotechnical Commission (IEC). The complete standard can be obtained from any ISO or IEC member or from the ISO or IEC Central Offices, Case Postal, 1211 Geneva 20, Switzerland. Copyright remains with ISO and IEC.

Portions of this book are extracted from *X/Open Specification, Programming Languages, Issue 4 Release 2*, copyright 1988, 1989, February 1992, by the X/Open Company Limited, with the permission of X/Open Company Limited. No further reproduction of this material is permitted without the written notice from the X/Open Company Ltd, UK.

Trademarks

The following terms are trademarks of International Business Machines Corporation in the United States or other countries or both:

AD/Cycle	AIX	AIX/6000
Application System/400	AS/400	BookManager
C Set ++	CICS	CICS/ESA
COBOL/370	C/MVS	C++/MVS
C/370	Common User Access	CUA
DATABASE 2	DB2	DFSMS
DFSMS/MVS	ESCON	GDDM
IBM	Hiperspace	IBMLink
IMS	IMS/ESA	Language Environment
MVS	MVS/DFP	MVS/ESA
MVS/SP	MVS/XA	Open Class
OpenEdition	OS/2	OS/390
OS/400	PROFS	PS/2
QMF	RACF	SAA
SOM	SOMobjects	SQL/DS
System/370	System Object Model	Systems Application Architecture
S/370	S/390	VM/ESA
VTAM	VisualAge	3090

Microsoft and Windows are trademarks or registered trademarks of Microsoft Corporation.

UNIX is a registered trademark in the United States and other countries licensed exclusively through X/Open Company Limited.

Other company, product, and service names, which may be denoted by a double asterisk(**), may be trademarks or service marks of others.

About this Book

This book provides guidance and reference information for the cross-language SOM collection classes along with instructions for building applications using these classes and the C++ SOM classes.

Required skills:

A cross language programmer who uses the SOM Cross-language Collection Classes should be familiar with programs that are written using the **Interface Definition Language** (IDL).

A Note on Samples:

The programming samples presented in this documentation are written in C language. A general cross language user must map these examples into the desired language.

IBM OS/390 C/C++ and Related Publications

This section summarizes the content of the IBM OS/390 C/C++ publications and shows where to find related information in other publications.

Table 1 (Page 1 of 3). OS/390 C/C++ Publications

Book Title and Number	Key Sections/Chapters in the Book
OS/390 C/C++ Programming Guide (1), SC09-2362	<ul style="list-style-type: none"> • C/C++ input and output • Debugging OS/390 C programs that use input/output • Using linkage specifications in C++ • Combining C and assembler • Creating and using DLLs • Using threads in an OS/390 OpenEdition application • Reentrancy • Using the decimal data type in C • Handling exceptions, error conditions and signals • Optimizing code • Optimizing your C/C++ code with Interprocedural Analysis • Network communications under OS/390 OpenEdition • Interprocess communications using OpenEdition Services • Structuring a program that uses C++ templates • Using environment variables • Using System Programming C facilities • Library functions for the System Programming C facilities • Using runtime user exits • Using the OS/390 C multitasking facility • Using other IBM products with OS/390 C/C++ (CICS, CSP, DWS, DB2, GDDM, IMS, ISPF, QMF) • Direct-to-SOM support under OS/390 C/C++ • Internationalization: locales and character sets, code set conversion utilities, mapping variant characters • POSIX character set • Code point mappings • Locales supplied with OS/390 C/C++ • Charmap files supplied with OS/390 C/C++ • Examples of charmap and locale definition source files • Converting code from code character set IBM-1047 • Using built-in functions • Programming considerations for OS/390 OpenEdition C/C++
OS/390 C/C++ User's Guide (1), SC09-2361	<ul style="list-style-type: none"> • OS/390 C/C++ examples • Compiler options • Binder options and control statements • Specifying OS/390 Language Environment runtime options • Compiling, IPA Linking, binding, and running OS/390 C/C++ programs • Using precompiled headers • Utilities (Object Library, DLL Rename, CXXFILT, DSECT Conversion, Code Set and Locale, ar and make, BPXBATCH) • Diagnosing problems • Cataloged procedures and REXX EXECs supplied by IBM • Error messages and return codes
OS/390 C/C++ Language Reference, SC09-2360	<ul style="list-style-type: none"> • Introduction to the C and C++ Languages • Lexical elements of OS/390 C and OS/390 C++ • Declarations, expressions and operators • Implicit type conversions • Functions and statements • Preprocessor directives • C++ classes, class members, and friends • C++ overloading, special member functions, and inheritance • C++ templates and exception handling • OS/390 C and OS/390 C++ compatibility

Table 1 (Page 2 of 3). OS/390 C/C++ Publications

Book Title and Number	Key Sections/Chapters in the Book
<i>OS/390 C/C++ Run-Time Library Reference</i> , SC28-1663	<ul style="list-style-type: none"> • Header files • Library functions
<i>OS/390 C Curses</i> , SC28-1907	<ul style="list-style-type: none"> • Overview of Curses <ul style="list-style-type: none"> – Key data types – General rules for characters, renditions, and window properties – General rules of operations and operating modes – Use of macros – Restrictions on block-mode terminals – Curses functional interface – Contents of headers – The terminfo database
<i>OS/390 C/C++ Compiler and Run-Time Migration Guide</i> , SC09-2359	<ul style="list-style-type: none"> • Common migration questions • Application executable program compatibility • Source program compatibility • Input and output operations compatibility • Class library migration considerations • Changes between releases of OS/390 • C/370 V1 to V2 compiler changes • Other migration considerations
<i>OS/390 C/C++ Reference Summary</i> , SX09-1313	<ul style="list-style-type: none"> • Summary tables of: <ul style="list-style-type: none"> – Character set, trigraphs, digraphs, and keywords – Escape sequences, storage classes – Predefined and derived types, type qualifiers – Operator precedence, redirection symbols – fprintf format, type characters, and flag characters – fscanf format and type characters – __amrc structure – Hardware exceptions and signals – Compiler return codes – Compiler options – #pragma directives – Library functions – Utilities
<i>OS/390 C/C++ IBM Open Class Library User's Guide</i> , SC09-2363	<ul style="list-style-type: none"> • Using the Complex Mathematics Class Library • Using the I/O Stream Class Library: Introduction, getting started, advanced topics, and manipulators • Using the Collection Class Library: Overview, instantiating and using, Element and Key functions, tailoring Collection implementation, polymorphic use of collections, support for notifications, exception handling, tutorials, problem solving, compatibility with previous releases, thread safety • Using the Application Support Class Library: Introduction, String classes, Exception and Trace classes, Date and Time classes, controlling threads and protecting data, the IBM Open Class notification framework, Binary Coded Decimal classes
<i>OS/390 C/C++ IBM Open Class Library Reference</i> , SC09-2364	<p>Reference information for:</p> <ul style="list-style-type: none"> • Complex Mathematics Class Library • I/O Stream Class Library • Collection Class Library • Application Support Class Library

Table 1 (Page 3 of 3). OS/390 C/C++ Publications

Book Title and Number	Key Sections/Chapters in the Book
<i>OS/390 C/C++ SOM-Enabled Class Library User's Guide and Reference</i> , SC09-2366	Guidance and reference information for: <ul style="list-style-type: none"> • C++ SOM (RRBC-enabled) versions of Collection and Application Support Class Libraries • Cross-language SOM version of the Collection Class Library
<i>Debug Tool User's Guide and Reference</i> , SC09-2137	<ul style="list-style-type: none"> • Preparing to debug programs • Debugging programs • Using Debug Tool in different environments • Language-specific information • Debug Tool reference
APAR and BOOKS files (Shipped with Program materials)	<ul style="list-style-type: none"> • Partitioned data set CBC.SCBCDOC on the product tape contains the members APAR and BOOKS which provide additional information for using the IBM OS/390 C/C++ licensed program, including <ul style="list-style-type: none"> – Isolating reportable problems – Keywords – Preparing Authorized Program Analysis Report (APAR) – Problem identification worksheet – Maintenance on OS/390 – Late changes to OS/390 C/C++ publications

Note:

1. For complete and detailed information on OS/390 Language Environment runtime options, linking, and running with OS/390 Language Environment, refer to the *OS/390 Language Environment Programming Guide*, SC28-1939. For complete and detailed information on using interlanguage calls, refer to *OS/390 Language Environment Writing Interlanguage Applications*, SC28-1943.

The following table lists the OS/390 C/C++ and related publications that you are most likely to need. Publications are grouped according to the tasks they describe.

Table 2 (Page 1 of 3). Publications by Task

Tasks	Books
Planning, preparing, and migrating to OS/390 C/C++	<p><i>OS/390 C/C++ Compiler and Run-Time Migration Guide</i>, SC09-2359</p> <p><i>OS/390 Language Environment Concepts Guide</i>, GC28-1945</p> <p><i>OS/390 Language Environment Customization</i>, SC28-1941</p> <p><i>OS/390 OpenEdition Introduction</i>, GC28-1889</p> <p><i>OS/390 OpenEdition XPG4 Conformance Document</i>, GC28-1897</p> <p><i>OS/390 Release 4 Planning for Installation</i>, GC28-1726 OS/390 Task Atlas, available on the OS/390 home page in the World Wide Web</p>
Installing	<p><i>OS/390 Program Directory</i></p> <p><i>OS/390 Release 4 Planning for Installation</i>, GC28-1726</p> <p><i>OS/390 Language Environment Customization</i>, SC28-1941</p>
Coding programs	<p><i>OS/390 C/C++ Run-Time Library Reference</i>, SC28-1663</p> <p><i>OS/390 C/C++ Language Reference</i>, SC09-2360</p> <p><i>OS/390 C/C++ Reference Summary</i>, SX09-1313</p> <p><i>OS/390 C/C++ Programming Guide</i>, SC09-2362</p> <p><i>OS/390 Language Environment Concepts Guide</i>, GC28-1945</p> <p><i>OS/390 Language Environment Programming Guide</i>, SC28-1939</p> <p><i>OS/390 Language Environment Programming Reference</i>, SC28-1940</p> <p><i>OS/390 C/C++ IBM Open Class Library User's Guide</i>, SC09-2363</p> <p><i>OS/390 C/C++ IBM Open Class Library Reference</i>, SC09-2364</p> <p><i>OS/390 C/C++ SOM-Enabled Class Library User's Guide and Reference</i>, SC09-2366</p>

Table 2 (Page 2 of 3). Publications by Task

Tasks	Books
Coding and binding programs with interlanguage calls	<p><i>OS/390 C/C++ Programming Guide</i>, SC09-2362</p> <p><i>OS/390 C/C++ Language Reference</i>, SC09-2360</p> <p><i>OS/390 Language Environment Programming Guide</i>, SC28-1939</p> <p><i>OS/390 Language Environment Writing Interlanguage Applications</i>, SC28-1943</p> <p><i>DFSMS/MVS Program Management</i>, SC28-1943</p>
Compiling, binding, and running programs	<p><i>OS/390 C/C++ User's Guide</i>, SC09-2361</p> <p><i>OS/390 Language Environment Programming Guide</i>, SC28-1939</p> <p><i>OS/390 Language Environment Debugging Guide and Run-Time Messages</i>, SC28-1942</p> <p><i>DFSMS/MVS Program Management</i>, SC26-4916</p> <p>OS/390 Messages Database, available from the OS/390 home page on the World Wide Web</p>
Compiling and binding applications in the OS/390 OpenEdition environment	<p><i>OS/390 C/C++ User's Guide</i>, SC09-2361</p> <p><i>OS/390 OpenEdition User's Guide</i>, SC28-1891</p> <p><i>OS/390 OpenEdition Command Reference</i>, SC28-1892</p> <p><i>DFSMS/MVS Program Management</i>, SC26-4916</p>
Compiling and binding SOM applications with OS/390 SOMobjects	<p><i>OS/390 SOMobjects Programmer's Guide</i>, GC28-1859</p> <p><i>OS/390 C/C++ Programming Guide</i>, SC09-2362</p> <p><i>OS/390 C/C++ User's Guide</i>, SC09-2361</p>
Debugging programs	<p>README file</p> <p><i>Debug Tool User's Guide and Reference</i>, SC09-2137</p> <p><i>OS/390 C/C++ User's Guide</i>, SC09-2361</p> <p><i>OS/390 C/C++ Programming Guide</i>, SC09-2362</p> <p><i>OS/390 Language Environment Programming Guide</i>, SC28-1939</p> <p><i>OS/390 Language Environment Debugging Guide and Run-Time Messages</i>, SC28-1942</p> <p><i>OS/390 OpenEdition Messages and Codes</i>, SC28-1908</p> <p><i>OS/390 OpenEdition User's Guide</i>, SC28-1891</p> <p><i>OS/390 OpenEdition Command Reference</i>, SC28-1892</p> <p><i>OS/390 OpenEdition Programming Tools</i>, SC28-1904</p>

Table 2 (Page 3 of 3). Publications by Task

Tasks	Books
Using shells and utilities in the OS/390 OpenEdition environment	<i>OS/390 C/C++ User's Guide</i> , SC09-2361 <i>OS/390 OpenEdition Command Reference</i> , SC28-1892 <i>OS/390 OpenEdition Messages and Codes</i> , SC28-1908
Using sockets library functions in the OS/390 OpenEdition environment	<i>OS/390 C/C++ Run-Time Library Reference</i> , SC28-1663
Performing diagnosis and submitting Authorized Program Analysis Report (APAR)	<i>OS/390 C/C++ User's Guide</i> , SC09-2361 CBC.SCBCDOC(APAR) on OS/390 C/C++ product tape
Quick reference	<i>OS/390 C/C++ Reference Summary</i> , SX09-1313
Multimedia Tutorial	For a new way of learning C++ programming, you can order the CD-ROM <i>Experience C++ A Multimedia Tutorial</i> , SK2T-1158. This tutorial runs in DOS.

Hardcopy Books

You can purchase OS/390 C/C++ books one at a time, or in a set. The following OS/390 C/C++ books are available in hardcopy:

- *OS/390 C/C++ Run-Time Library Reference*, SC28-1663
- *OS/390 C/C++ User's Guide*, SC09-2361
- *OS/390 C/C++ Programming Guide*, SC09-2362
- *OS/390 C/C++ Reference Summary*, SX09-1313
- *OS/390 C/C++ IBM Open Class Library User's Guide*, SC09-2363
- *OS/390 C Curses*, SC28-1907
- *OS/390 C/C++ Compiler and Run-Time Migration Guide*, SC09-2359
- *Debug Tool User's Guide and Reference*, SC09-2137

These books can be purchased singly or as part of a set. The *OS/390 C/C++ Compiler and Run-Time Migration Guide*, SC09-2359 is provided at no charge. The remaining books are included in feature code 8009.

Softcopy Books

All of the OS/390 C/C++ publications (except for the *OS/390 C/C++ Reference Summary*) are available in softcopy book format. The books are available on a tape accompanying the OS/390 product, and also on a CD-ROM called the *IBM Online Library Omnibus Edition: OS/390 Collection*, SK2T-6700.

To read the softcopy books, the BookManager Read (Program 5684-062, 5695-046) licensed program must be available on your operating system. BookManager Read provides access to online information as an alternative to hard copy documents. You can read, search, make notes, and select sections of text to print.

Also available are BookManager Read/DOS (Program 73F6-022) for the DOS operating system, and BookManager Read/2 (Program 73F6-023) for the OS/2 operating system. With these products, you can download online books to your workstation and read them.

With BookManager Read installed on your system, you can enter the command BOOKMGR to start BookManager and display a list of books available to you. If you know the name of the book that you want to view, you can use the OPEN command to open the book directly.

Note: If your workstation does not have graphics capability, BookManager Read cannot correctly display some characters, such as arrows and brackets.

You can also browse the books on the World Wide Web, through "The Library" link on the OS/390 home page. The URL for this page is:

<http://www.s390.ibm.com/os390/index.html>

Softcopy Examples

Most of the larger examples in the following books are available in machine-readable form:

- *OS/390 C/C++ Language Reference*, SC09-2360
- *OS/390 C/C++ User's Guide*, SC09-2361
- *OS/390 C/C++ Programming Guide*, SC09-2362
- *OS/390 C/C++ IBM Open Class Library User's Guide*, SC09-2363
- *OS/390 C/C++ IBM Open Class Library Reference*, SC09-2364
- *OS/390 C/C++ SOM-Enabled Class Library User's Guide and Reference*, SC09-2366

Softcopy examples are indicated in the book by a label. In the following books, the label has the form CBCxyyy or CLBxyyy, where x refers to a publication:

- R and X refer to the *OS/390 C/C++ Language Reference*
- G refers to the *OS/390 C/C++ Programming Guide*
- U refers to the *OS/390 C/C++ User's Guide*
- A refers to the *OS/390 C/C++ IBM Open Class Library User's Guide*

An exception applies to the Collection Class Library example names, which do not follow a naming convention. These examples are in the *OS/390 C/C++ IBM Open Class Library Reference* and in the *OS/390 C/C++ SOM-Enabled Class Library User's Guide and Reference*.

For all books other than the *OS/390 C/C++ SOM-Enabled Class Library User's Guide and Reference*, the label refers to a member name in the data set *CBC.SCBCSAM* or *CBC.SCLBSAM*. For the *OS/390 C/C++ SOM-Enabled Class Library User's Guide and Reference*, the label refers to a member name in the data set *CBC.SCLBXSAM*.

For Late Breaking C/C++ News...

IBM also publishes the *C/370 Compiler Newsletter*. This free newsletter keeps subscribers up to date on the latest product releases, provides coding hints and tips, questions and answers, and news about C/370 products and IBM OS/390 C/C++.

To take advantage of this free publication, send your name, full mailing address, and phone number (in case we have to talk to you), in one of these ways:

- Send a message electronically to the following network ID :

- Internet: `inetc370@vnet.ibm.com`
- IBMMAIL: `ibmmail(caibmrzx)`

- Mail your request to:

EDITOR, C/370 Compiler Newsletter
IBM Canada Ltd. Laboratory
9/604/895/TOR
895 Don Mills Road
NORTH YORK ONTARIO CANADA M3C 1W3

OS/390 C/C++ on the World Wide Web

Additional information on OS/390 C/C++ is available on the World Wide Web. The URL for the OS/390 C/C++ home page is:

<http://www.software.ibm.com/ad/c370/>

About IBM OS/390 C/C++

The C/C++ feature of the IBM OS/390 licensed program provides support for C and C++ application development on the OS/390 platform. The C/C++ feature is based on the C/C++ for MVS/ESA product.

IBM OS/390 C/C++ includes:

- A C compiler (referred to as the OS/390 C compiler)
- A C++ compiler (referred to as the OS/390 C++ compiler)
- A set of C++ class libraries
- Application Support Class and Collection Class Library source
- A mainframe interactive Debug Tool (optional)
- A set of C/C++ application development utilities

IBM offers the C language on other platforms, such as the AIX, OS/2, OS/400, Sun** Solaris**, VM/ESA, VSE, and Windows** operating systems. The C++ language is also offered on the AIX, OS/2, OS/400, Sun Solaris, and Windows operating systems.

The C/C++ Compilers

The following sections describe the C and C++ languages and the OS/390 C/C++ compilers.

The C Language

The C language is a general purpose, versatile, function-oriented programming language that allows a programmer to create applications quickly and easily. C provides high-level control statements and data types as do other structured programming languages, and it also provides many of the benefits of a low-level language.

The C++ Language

The C++ language introduces object-oriented concepts into the C language, on which it is based. For a detailed description of the differences between OS/390 C++ and OS/390 C, refer to the *OS/390 C/C++ Language Reference*.

The C++ language introduces classes, which are user-defined data types that may contain both data and function definitions. This ability to define both functions and data is data abstraction. You can use classes from established class libraries, develop your own classes, or derive new classes from existing classes by adding data descriptions and functions. New classes can inherit properties from one or more classes. Not only do classes describe the data types and functions available, but they can also hide (encapsulate) the implementation details from user programs. An object is an instance of a class.

The C++ language also provides templates and other features including access control to data and functions, and better type checking and exception handling than the C language. It also supports polymorphism and operator overloading.

Common Features of the OS/390 C and C++ Compilers

The OS/390 C/C++ compilers provide you with features such as the following:

- Optimization support through the following facilities:
 - Algorithms to take advantage of S/390 architecture to get better optimization for speed and use of computer resources
 - The OPTIMIZE compile-time option to instruct the compiler to optimize the machine instructions it generates, to produce faster running object code
 - Interprocedural Analysis (IPA), to perform optimizations across compilation units
 - The precompiled header facility, to save information from one compilation unit for use in another

- DLLs (Dynamic Link Libraries) to reduce application size and to provide load-on-reference support.

IBM OS/390 C/C++ provides support for generating DLLs in a way similar to the way OS/2 DLLs are generated. DLLs allow a function reference or a variable reference in one executable to use a definition located in another executable at run time. You can use both load-on-reference and load-on-demand DLLs. A load-on-reference DLL is made available when a DLL function is called or a DLL variable is referenced. Load-on-demand DLLs are explicitly controlled by the application code at the source level.

You can use DLLs to split applications into smaller modules and improve system memory usage. DLLs also offer more flexibility for building, packaging, and redistributing applications.

- Full program reentrancy.

With reentrancy, a program can be run simultaneously by many users. A reentrant program that is stored in the LPA (Link Pack Area) or ELPA (Extended Link Pack Area) and is run by multiple users simultaneously uses less storage, reduces processor I/O when the program starts up, and improves program performance by reducing the transfer of data to auxiliary storage. OS/390 C programmers can design programs that are naturally reentrant. For those programs that are not naturally reentrant, C programmers can use constructed reentrancy by compiling the programs with the RENT option and using the program management binder supplied with OS/390, or the OS/390 Language Environment Prelinker (prelinker) and the Linkage Editor, to make them reentrant. The OS/390 C++ compiler always ensures that C++ programs are reentrant.

- Locale-based internationalization support derived from the IEEE POSIX** 1003.2-1992 standard and from the X/Open** CAE Specification, System Interface Definitions, Issue 4 and Issue 4 Version 2. This allows programmers to use locales to specify language/country characteristics for their applications.
- The ability to call and be called by other languages such as assembler, COBOL, PL/1, and Fortran, to enable programmers to integrate OS/390 C/C++ code with existing applications.
- Exploitation of OS/390 and OS/390 OpenEdition technology.

OS/390 OpenEdition is an IBM implementation of the open operating system environment, as defined in the XPG4 and POSIX standards.

- When used with OpenEdition Services and OS/390 Language Environment, support for the following standards at the system level:
 - A subset of the extended multibyte and wide character functions as defined by the Programming Language C Amendment 1, which is ISO/IEC 9899:1990/Amendment 1:1994(E)
 - ISO/IEC 9945-1:1990(E)/IEEE POSIX 1003.1-1990
 - A subset of IEEE POSIX 1003.1a, Draft 6, July 1991
 - IEEE Portable Operating System Interface (POSIX) Part 2, P1003.2
 - A subset of IEEE POSIX 1003.4a, Draft 6, February 1992 (POSIX.4a has been renumbered by the IEEE POSIX committee to POSIX.1c)
 - X/Open CAE Specification, System Interfaces and Headers, Issue 4 Version 2
 - X/Open CAE Specification, Network Services, Issue 4
- Year 2000 support.

Features Specific to the OS/390 C Compiler

In addition to the features common to OS/390 C/C++, the OS/390 C compiler provides you with the following:

- The ability to write portable code conforming to the following standards:
 - All elements of the ISO standard ISO/IEC 9899:1990 (E)
 - ANSI/ISO 9899:1990[1992] (formerly ANSI X3.159-1989 C)
 - X/Open Specification Programming Language Issue 3, Common Usage C
 - FIPS-160
- System programming capabilities, which allow you to use OS/390 C in place of assembler
- Additional optimization capabilities through the `INLINE` compile-time option
- Extensions of the standard C/C++ language definitions to provide programmers with support for the OS/390 environment, such as fixed-point (packed) decimal data support

Features Specific to the OS/390 C++ Compiler

In addition to the features common to OS/390 C/C++, the OS/390 C++ compiler provides you with the following:

- An implementation based on the definition of the language contained in the Draft Proposal International Standard for Information Systems– Programming Language C++ (X3J16/92-00091). The OS/390 C++ compiler also conforms to a subset of the C++ ANSI/ISO (Draft) Standard (X3J16/93-0062).
- System Object Model (SOM) support, through the SOM Interface Definition Language (IDL) compiler available with OS/390 SOMobjects. The IDL compiler and associated emitters can be used to create language-specific bindings that define the interface to a SOM object. This enables OS/390 C/C++ programs to share SOM objects with other languages. In addition, SOM enables release-to-release binary compatibility.

With Direct-to-SOM (DTS) support in the OS/390 C++ compiler, you can generate SOM objects directly from C++ code. You do not need to create and process IDL first. You can write virtually the same code you do when creating C++ objects.

Note: The OS/390 C/C++ compiler no longer supports IDL generation through the IDL compile-time option. This option instructed the compiler to generate IDL, which is required for mixed-language or distributed object applications. If you need IDL for your applications, you should now code it yourself instead of generating it through the IDL compile option.

- C++ template support and exception handling consistent with VisualAge C++ product implementations.

Utilities

The following utilities are provided with the OS/390 C/C++ compilers:

- The Object Library Utility to update PDS libraries of object modules and IPA object modules
- The DLL Rename Utility to make selected DLLs a unique component of the applications with which they are packaged
- The CXXFILT Utility to map OS/390 C++ mangled names to the original source
- The localedef Utility to read the locale definition file and produce a locale object that the locale-specific library functions can use
- The DSECT Conversion Utility to convert descriptive assembler DSECTs into OS/390 C/C++ data structures
- The C/C++ Model Tool to provide online help for C/C++ `#pragma` directives and runtime library functions (other than the C Curses functions) at the level supplied in OS/390 Release 2

Class Libraries

IBM OS/390 C/C++ provides a base set of class libraries, called IBM Open Class, which is consistent with that available in other members of the VisualAge C++ product family. These class libraries are:

- The I/O Stream Class Library

The I/O Stream Class Library lets you view input and output (I/O) independent of physical I/O devices or of data types used. You can code sophisticated I/O statements easily and clearly, and define input and output for your own data types. By using the I/O Stream Class Library, you can improve the maintainability of programs that use input and output.

- The Complex Mathematics Class Library

The Complex Mathematics Class Library lets you manipulate and perform standard arithmetic on complex numbers. Complex numbers are used in the scientific and technical fields.

- The Application Support Class Library

The Application Support Class Library provides the basic abstractions needed during the creation of most C++ applications, including String, Date, and Time.

The Application Support Class library is available in a C++ SOM as well as the regular C++ native version.

- The Collection Class Library

The Collection Class Library uses data abstraction to implement a wide variety of classical data structures such as stack, tree, list, hash table, and so on. Collections are used by most programs. Programs can be developed without having to define every collection. Programmers can start programming using a high level of abstraction and later replace an abstract data type with the appropriate concrete implementation. Each abstract data type has a common interface for all of its implementations. The Collection Class Library provides programmers with a consistent set of building blocks from which application objects can be derived. The library is designed to exploit such C++ language features as exception handling and template support.

The Collection Class Library is available in a C++ SOM and a cross-language SOM version, as well as the regular C++ native version.

All of the libraries described above, except the cross-language SOM version of the Collection Class Library, are thread-safe.

All of the libraries described above are available in both static and DLL formats. The Application Support Class and Collection Class libraries are packaged together in a single DLL. For compatibility, separate side-decks are available for the Application Support Class and Collection Class libraries, in addition to the side-deck available for the combined library.

Note: If your product uses the IBM-supplied DLLs, you must use the DLL Rename utility to rename them, and you must ship those renamed DLLs with your product. Refer to the *OS/390 C/C++ User's Guide* for information about the DLL Rename utility.

Class Library Source

The Class Library Source consists of the following:

- Application Support Class Library source code
- Collection Class Library source code (C++ native and C++ SOM only)
- Instructions for building the Application Support Class and Collection Class Libraries in C++ native (static and DLL) versions
- Instructions for building the Application Support Class and Collection Class Libraries in C++ SOM (static and DLL) versions
- Class Library Language Environment message file source
- Instructions for building the Class Library Language Environment message files

The Debug Tool

IBM OS/390 C/C++ supports program development using a mainframe interactive Debug Tool. This optionally available tool allows you to debug applications in their native host environment, such as CICS/ESA, IMS/ESA, DB2, and so on. The Debug Tool provides the following support and function:

- Step mode
- Breakpoints

- Monitor
- Frequency analysis
- Dynamic patching

The debug session can be recorded in a log file, so you can replay the session. You can use the Debug Tool to help capture test cases for future program validation or to further isolate a problem within an application.

You can specify either data sets or UNIX-like hierarchical file system (HFS) files as source files.

OS/390 Language Environment

IBM OS/390 C/C++ exploits the C/C++ runtime environment and library of runtime services available with OS/390 Language Environment (formerly Language Environment for MVS & VM, Language Environment/370 and LE/370).

OS/390 Language Environment consists of four language-specific runtime libraries, and Base Routines and Common Services, as shown in Figure 1. OS/390 Language Environment establishes a common runtime environment and common runtime services for language products, user programs, and other products.

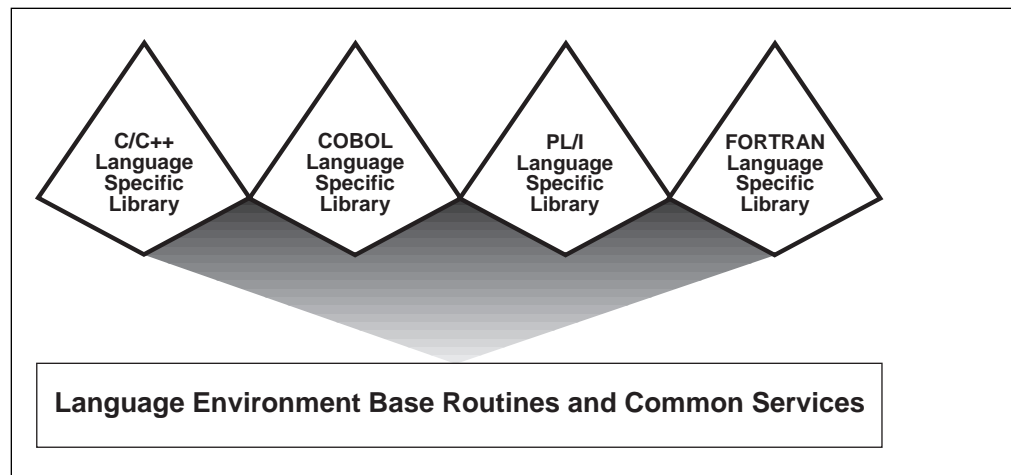


Figure 1. Libraries in OS/390 Language Environment

The common execution environment is made up of data items and services included in library routines available to an application running in the environment. The services that OS/390 Language Environment provides include:

- Services that satisfy basic requirements common to most applications. These include support for the initialization and termination of applications, allocation of storage, interlanguage communication (ILC), and condition handling.
- Extended services often needed by applications. These functions are contained within a library of callable routines, and include interfaces to operating system functions and a variety of other commonly used functions.
- Runtime options that help in the execution, performance, and diagnosis of your application.

- Access to operating system services; OS/390 OpenEdition services are available to an application programmer or program through the OS/390 C/C++ language bindings.
- Access to language-specific library routines, such as the OS/390 C/C++ library functions.

The binder provided with OS/390 combines the object modules, load modules, and program objects comprising an OS/390 application to produce a single output program object that you can then load for execution. The binder supports all C and C++ code, provided that the output program is stored in a PDSE (Partitioned Data Set Extended) member or an HFS file.

If you cannot use a PDSE member or HFS file, and your program contains C++ code, or C code that is compiled with any of the RENT, LONGNAME, DLL or IPA compile-time options, you must use the prelinker.

Using the binder without using the prelinker has the following advantages:

- Faster rebinds when recompiling and rebinding a few of your source files
- Rebinding at the single compile unit level of granularity (except when you use the IPA compile-time option)
- Input of object modules, load modules, and program objects
- Improved long name support:
 - Long names do not get converted into prelinker generated names
 - Long names appear in the binder maps, enabling full cross-referencing/li>
 - Variables do not disappear after prelink
 - Fewer steps in the process of producing your executable program

The prelinker provided with OS/390 Language Environment combines the object modules comprising an OS/390 C/C++ application and produces a single object module that you can link-edit into a load module (which is stored in a PDS), or bind into a load module or a program object stored in a PDS, or a PDSE or HFS file.

OS/390 OpenEdition

OS/390 OpenEdition provides capabilities under OS/390 to make it easier to implement or port applications in an open, distributed environment.

OS/390 OpenEdition Services

OS/390 OpenEdition Services are available to OS/390 C/C++ application programs through the C/C++ language bindings available with OS/390 Language Environment.

Together, the OS/390 OpenEdition Services, OS/390 Language Environment, and OS/390 C/C++ compilers provide an application programming interface that supports industry standards.

OS/390 OpenEdition support for both existing OS/390 applications and new OpenEdition applications includes:

- C programming language support as defined by ISO/ANSI C
- C++ programming language support

- C language bindings as defined in: the IEEE 1003.1 and 1003.2 standards, subsets of the draft 1003.1a and 1003.4a standards, X/Open CAE Specification: System Interfaces and Headers, Issue 4, Version 2, which provides standard interfaces for better source code portability with other conforming systems, and X/Open CAE Specification, Network Services, Issue 4, which defines the X/Open UNIX** descriptions of sockets and X/Open Transport Interface (XTI)
- OS/390 OpenEdition Extensions that provide OS/390-specific support beyond the defined standards
- The OS/390 OpenEdition Shell and Utilities feature, which provides:
 - A UNIX-like user interface (with support for POSIX.2 and XPG4.2)
 - Support for tools and utilities as defined in the CAE Specification, Commands and Utilities, Issue 4 (XPG4), as well as OS/390 support, including the following:

ar	Creates and maintains library archives
BPXBATCH	Allows you to submit batch jobs that run shell commands or scripts or OS/390 C/C++ executable files in HFS files from a shell session
c89	Compiles and link-edits OS/390 C applications
gencat	Merges the message text source files Messagefile (usually *.msg) into a formatted message Catalogfile (usually *.cat)
lex	Writes large parts of a lexical analyzer automatically, based on a description supplied by the programmer
make	Helps you manage projects containing a set of interdependent files, such as a program with many OS/390 C/C++ source and object files, by keeping all such files up to date with one another
yacc	Allows you to write compilers and other programs that parse input according to strict grammar rules
 - Support for other utilities such as:

c++	Compiles, binds, prelinks and link-edits OS/390 OpenEdition C++ applications
mkcatdefs	Preprocesses a message source file for input to the gencat utility
runcat	Invokes mkcatdefs and pipes the message catalog source data (the output from mkcatdefs) to gencat
dspcat	Displays all or part of a message catalog
dspmsg	Displays a selected message from a message catalog
- The OpenEdition Debugger feature, which provides the dbx interactive symbolic debugger for OS/390 OpenEdition applications
- OS/390 OpenEdition services that provide access to a UNIX-like hierarchical file system (HFS), with support for the POSIX.1 and XPG4 standards
- OS/390 C/C++ I/O routines support using HFS files, standard OS/390 data sets, or a mixture of both

- Application threads (with support for a subset of POSIX.4a)
- Support for OS/390 C/C++ DLLs

OS/390 OpenEdition offers program portability across multivendor operating systems, with support for POSIX.1, POSIX.1a (draft 6), POSIX.2, POSIX.4a (draft 6), and XPG4.2.

To application developers who have used UNIX-like environments, the OS/390 OpenEdition Shell and Utilities are a familiar C/C++ application development environment. If you are familiar with existing MVS development environments, you may find that the OS/390 OpenEdition environment can enhance your productivity. Refer to the *OS/390 OpenEdition User's Guide* for more information on the Shell and Utilities.

OS/390 C/C++ Applications with OpenEdition C/C++ Functions

Most OpenEdition C functions are available at all times. However, to use some OpenEdition C functions, an OS/390 C/C++ program must be run on a system with the OS/390 OpenEdition kernel available and active. In some situations, you must also specify the `POSIX(ON)` runtime option. This is required for the POSIX.4a threading functions, and for such functions as the system and signal handling functions, where the behavior is different between POSIX/XPG4 and ANSI. Refer to the *OS/390 C/C++ Run-Time Library Reference* for more information about requirements for each function.

Some of the ways an OS/390 C/C++ program that uses OpenEdition C functions can be invoked are:

- The program can be invoked directly from the OS/390 OpenEdition Shell.
- The program can be invoked from another program, or from the OS/390 OpenEdition Shell, using one of the `exec` family of functions, or the `BPXBATCH` utility from TSO or MVS batch.
- The program can be invoked using the POSIX **system** call.
- The program can be invoked directly through TSO or MVS batch without the use of the intermediate `BPXBATCH` utility. In some cases, the `POSIX(ON)` runtime option is required.

Input and Output

The C/C++ runtime library that supports the OS/390 C/C++ compiler supports different input and output (I/O) interfaces, file types, and access methods. Additional support is provided by the C++ I/O Stream Class Library.

I/O Interfaces

The C/C++ runtime library supports the following I/O interfaces:

C Stream I/O

This is the default and the ANSI-defined I/O method. All I/O is processed by character.

Record I/O

Your input and output can also be processed by record. A record is a set of data treated as a unit. Record processing of VSAM data sets is also supported.

Record I/O is an OS/390 C/C++ extension to the ANSI standard.

TCP/IP Sockets I/O

OS/390 OpenEdition provides support for an enhanced version of an industry-accepted protocol for client/server communication known as *sockets*. A set of C language functions provides support for OS/390 OpenEdition sockets. OS/390 OpenEdition sockets correspond closely to the sockets used by UNIX applications that use the Berkeley Software Distribution (BSD) 4.3 standard (also known as OE sockets). The slightly different interface of the X/Open CAE Specification, Networking Services, Issue 4, is supplied as an additional choice. This interface is known as X/Open Sockets.

The OS/390 OpenEdition socket application program interface (API) provides support for both UNIX domain sockets and Internet domain sockets. UNIX domain sockets, or *local sockets*, allow interprocess communication within OS/390 independent of TCP/IP. Local sockets behave like traditional UNIX sockets and allow processes to communicate with one another on a single system. Internet sockets allow application programs to communicate with others in the network using TCP/IP.

In addition, the C++ I/O Stream Library supports formatted I/O in C++. You can code sophisticated I/O statements easily and clearly, and define input and output for your own data types. This helps improve the maintainability of programs that use input and output.

File Types

In addition to conventional files, such as sequential files and partitioned data sets, the C/C++ runtime library supports the following file types:

Virtual Storage Access Method (VSAM) Data Sets

OS/390 C/C++ has native support for three types of VSAM data organization:

- Key-sequenced data sets (KSDS) are used when a record will normally be accessed through a key within the record. A key is one or more consecutive characters taken from a data record that identifies the record.
- Entry-sequenced data sets (ESDS) are used when data will be accessed in the order it was created (or the reverse order).
- Relative-record data sets (RRDS) are used for data in which each item has a particular number (for example, a telephone system with a record associated with each number).

All three types are ordered and can have keys associated with their records.

Hierarchical File System Files

When you are running under MVS, TSO (batch and interactive), or IMS, OS/390 C/C++ recognizes a Hierarchical File System (HFS) file as such if the name specified on the `fopen()` or `freopen()` call conforms to certain rules (described in the *OS/390 C/C++ Programming Guide*). You can create regular, link, directory, character special, or FIFO HFS files.

Memory Files

Memory files are temporary files that reside in memory. For improved performance, you can direct input and output to memory files rather than to devices. Since memory files reside in main storage and only exist while the program is executing, they are primarily used as work files. Memory files can be accessed across load modules through calls to non-POSIX `system()` and `C fetch()`; they exist for the life of the root program. Standard streams can be redirected to memory files on a non-POSIX `system()` call using command line redirection.

Hiperspace Expanded Storage

Large memory files can be placed in Hiperspace expanded storage to free up some of your home address space for other uses. Hiperspace expanded storage or high performance space is a range of up to 2 gigabytes of contiguous virtual storage space that a program can use as a buffer (1 gigabyte = 2^{30} bytes).

Additional I/O Features

IBM OS/390 C/C++ provides additional I/O support through the following features:

- User error handling for serious I/O failures (SIGIOERR)
- Improved sequential data access performance through enablement of the DFSMS/MVS support for 31-bit sequential data buffers and sequential data striping on extended format data sets
- Full support of PDS/Es on OS/390, including support for multiple members opened for write
- Overlapped I/O support under OS/390 (NCP, BUFNO)
- Multibyte character I/O functions
- Fixed-point (packed) decimal data type support in formatted I/O functions
- Support for multiple volume data sets that span more than one volume of DASD or tape
- Support for Generation Data Group I/O

The System Programming C Facility

The System Programming C (SPC) facility allows you to build applications that require no dynamic loading of OS/390 Language Environment libraries, and allows you to tailor your application to better utilize the low-level services available on your operating system. You can:

- Develop SPC applications that can be executed in a customized environment rather than with OS/390 Language Environment services. Note that if you do not use OS/390 Language Environment services, only some built-in functions and a limited set of C/C++ runtime library functions are available to you.
- Use the OS/390 C language as an assembler language substitute for writing system exit routines, with the interfaces provided by SPC.
- Develop applications featuring:
 - A user-controlled environment, in which an OS/390 C environment is created once and used repeatedly for C function execution from other languages.
 - Co-routines using a two-stack model to write application service routines. In this model, the application calls on the service routine to perform services independently of the user, and then is suspended when control is returned to the user application.

Interaction with Other IBM Products

When you use OS/390 C/C++, you can write programs that take advantage of the power of other IBM products and subsystems. These products and subsystems are:

- Cross System Product (CSP)

Cross System Product/Application Development (CSP/AD) is an application generator that provides ways to interactively define, test, and generate application programs to improve productivity in application development. Cross System Product/Application Execution (CSP/AE) takes the generated program and executes it in a production environment.

Note: CSP applications cannot be compiled with the OS/390 C++ compiler. But your OS/390 C++ program can use ILC to call OS/390 C programs that access CSP.

- Customer Information Control System (CICS)

You can use the CICS/ESA Command-Level Interface to write C/C++ application programs. The CICS Command-Level Interface provides data, job, and task management facilities that are normally provided by the operating system.

Note: Code preprocessed with CICS/ESA versions prior to V4 R1 is not supported for OS/390 C++ applications. OS/390 C++ code preprocessed on CICS/ESA V4 R1 cannot run under CICS/ESA V3 R3.

- DATABASE 2 (DB2)

DB2 programs manage data stored in relational data bases. The IBM DATABASE 2 Licensed Program runs on OS/390.

You can access the data using a structured set of queries written in a language called Structured Query Language (SQL). The DB2 program uses SQL

statements imbedded in the program. The SQL translator (DB2 preprocessor) translates the imbedded SQL into host language statements that perform the requested functions. The OS/390 C/C++ compilers compile the output of the SQL translator. The DB2 program processes a request and processing returns to the application.

- Data Window Services (DWS)

The Data Window Services (DWS) part of the Callable Services Library allows your OS/390 C or OS/390 C++ program to manipulate temporary data objects known as TEMPSPACE and VSAM linear data sets.

- Information Management System (IMS)

The Information Management System/Enterprise Systems Architecture (IMS/ESA) product provides support for hierarchical databases.

- Interactive System Productivity Facility (ISPF)

OS/390 C/C++ provides access to the Interactive System Productivity Facility (ISPF) Dialog Management Services. A dialog is the interaction between a person and a computer. The dialog interface contains display, variable, message, and dialog services as well as other facilities that are used to write interactive applications.

- Graphical Data Display Manager (GDDM)

GDDM provides a comprehensive set of functions to display and print applications most effectively. These functions include:

- A windowing system that the user can tailor to display selected information
- Support for presentation and keyboard interaction
- Comprehensive graphics support
- Fonts, including support for double-byte character set (DBCS)
- Business image support
- Saving and restoring graphics pictures
- Support for many types of display terminals, printers and plotters

- Query Management Facility (QMF)

OS/390 C supports the Query Management Facility (QMF), a query and report writing facility, which allows you to write applications through a callable interface. You can create applications to perform a variety of tasks, such as data entry, query building, administration aids, and report analysis.

Additional Features of OS/390 C/C++

Feature	Description
Multibyte Character Support	OS/390 C/C++ supports multibyte characters for those national languages such as Japanese whose characters cannot be represented by a single byte.

Feature	Description
Wide Character Support	Multibyte characters can be normalized by OS/390 C library functions and encoded in units of one length. These normalized characters are called wide characters. Conversions between multibyte and wide characters can be performed by string conversion functions such as <code>wcstombs</code> , <code>mbstowcs</code> , <code>wcsrtombs</code> , and <code>mbsrtowcs</code> , as well as the family of wide-character I/O functions. Wide-character data can be represented by the <code>wchar_t</code> data type.
Extended Precision Floating-Point Numbers	OS/390 C/C++ provides three S/370 floating-point number data types: single precision (32 bits), declared as <code>float</code> ; double precision (64 bits), declared as <code>double</code> ; and extended precision (128 bits), declared as <code>long double</code> . Extended precision floating-point numbers give greater accuracy to mathematical calculations.
Command Line Redirection	You can redirect the standard streams <code>stdin</code> , <code>stderr</code> , and <code>stdout</code> from the command line or when calling programs using the <code>system()</code> function.
National Language Support	OS/390 C/C++ provides message text in either American English or Japanese. You can dynamically switch between the two languages.
Locale Definition Support	OS/390 C/C++ provides a locale definition utility that supports the creation of separate files of internationalization data, or locales. Locales can be used at run time to customize the behavior of an application to national language, culture, and coded character set (code page) requirements. Locale-sensitive library functions, such as <code>isdigit</code> , use this information.
Coded Character Set (Code page) Support	The OS/390 C/C++ compiler can compile C/C++ source written in different EBCDIC code pages. In addition, the <code>iconv</code> utility converts data or source from one code page to another.
Selected Built-in Library Functions	Selected library functions, such as string and character functions, are built into the compiler to improve performance execution. Built-in functions are compiled into the executable, and no calls to the library are generated.
Multitasking Facility (MTF)	Multitasking is a mode of operation where your program performs two or more tasks at the same time. OS/390 C provides a set of library functions that perform multitasking. These functions are known as the Multitasking Facility (MTF). MTF uses the multitasking capabilities of OS/390 to allow a single OS/390 C application program to use more than one processor of a multiprocessing system simultaneously.
Packed Structures and Unions	OS/390 C provides support for packed structures and unions. Structures and unions may be packed to reduce the storage requirements of a OS/390 C program.
Fixed-point (Packed) Decimal Data	OS/390 C supports fixed-point (packed) decimal as a native data type for use in business applications. The packed data type is similar to the COBOL data type <code>COMP-3</code> or the PL/I data type <code>FIXED DEC</code> , with up to 31 digits of precision. The Application Support Class Library provides the Binary Coded Decimal Class for C++ programs.
Long Name Support	For portability, external names can be mixed case and up to 1024 characters in length. For C++, the limit applies to the mangled version of the name.
System Calls	You can call commands or executable modules using the <code>system</code> function under OS/390, OS/390 OpenEdition services, and TSO. You can also use the <code>system</code> function to call EXECs on OS/390 and TSO, or Shell scripts using OS/390 OpenEdition services.
Exploitation of ESA	Support for OS/390, IMS/ESA, Hiperspace expanded storage, and CICS/ESA allows you to exploit the features of the ESA.

Feature	Description
Exploitation of hardware	ARCH(2) instructs the compiler to generate faster instruction sequences available only on newer machines. TUNE(2) allows the executable program to be optimized for a certain architecture. For information on which machines and architectures support the above options, refer to the ARCHITECTURE and TUNE compiler information in the <i>OS/390 C/C++ User's Guide</i> .

C++SOM and Cross-language SOM Class Libraries

Chapter 1. C++ SOM and Cross-language SOM Class Libraries	5
SOM-enabled and Not SOM-enabled Versions	5
C++ SOM and Cross-language SOM Collection Classes	6
Coding with Class Libraries under OS/390 OpenEdition Services	7
Compiling and Binding with the C++ SOM Libraries	7
Using the Cross-language SOM Collection Class Library	9

Chapter 1. C++ SOM and Cross-language SOM Class Libraries

This release includes three types of class libraries:

- C++ native
- C++ SOM, which provides release-to-release binary compatibility (RRBC)
- Cross-language SOM, which provides RRBC and cross-language support

The I/O Stream and Complex Mathematics Class Libraries are available in C++ native versions only.

The Application Support Class Library is available in:

- C++ native
- C++ SOM

The Collection Class Library is available in:

- C++ native
- C++ SOM
- Cross-language SOM

All of these class libraries are available in both static and DLL forms.

The C++ native versions of the libraries are documented in the *OS/390 C/C++ IBM Open Class Library User's Guide* and *OS/390 C/C++ IBM Open Class Library Reference*.

You use the C++ SOM class libraries the same way as the C++ native versions, except that you compile and bind them differently. Instructions for compiling and binding the C++ SOM class libraries are described in this book; for guidance and reference information on these libraries, see the books listed above for C++ native class libraries.

All information for the cross-language SOM Collection Class Library is included in this book.

SOM-enabled and Not SOM-enabled Versions

When you use the SOM-enabled versions of the library, which provide RRBC, your program can run with future releases of the library without needing to be recompiled. Recompile is normally required whenever either your program or the library changes; with the RRBC-enabled library, you normally do not need to recompile unless your program changes. Depending on the RRBC version of the library you use (DLL or static) you may need to rebind your application:

- If you used the RRBC DLL you neither need to recompile nor rebind your application.
- If you used the static library you will only have to rebind your application with the new static RRBC library.

Release-to-release binary compatibility for the library is achieved by using the Direct-To-SOM support provided by the OS/390 C/C++ compiler. All classes in the

library accessible by applications inherit from SOMObject, and method resolution is performed using the SOM method resolution mechanism. For details on Direct-To-SOM support, see "Direct-to-SOM Support under OS/390 C/C++" in the *OS/390 C/C++ Programming Guide*.

Although RRBC can help you reduce development effort and make it easier to redistribute your programs, there are performance and code size trade-offs you should be aware of. Adding SOM-enablement to a program tends to increase its code size, whether the program uses the DLL version or the static version of a library. The RRBC version of the DLL is also larger than the C++ native version, although DLL size does not have a significant impact on DLL load-time. Finally, method resolution via SOM is slower than pure C++ method resolution. Therefore, for reasons of performance or code size, you may want to use the C++ native versions of the library as these are not RRBC-enabled. With these versions, you may have to recompile your application with a possible future version of the library, because these versions do not guarantee release-to-release binary compatibility.

Why Multiple Library Versions?

There are good reasons for using each of the library versions. The following table has a column for each RRBC/non-RRBC library version, and rows for different objectives. A single X in a cell indicates that the library version accomplishes the objective; however, a double X means that it accomplishes the objective better than other library versions. To determine what RRBC/non-RRBC library version you should use, identify the objectives that are most important to you, and look for a column where each of the objectives has a X. Some objectives, such as reducing code size and eliminating recompilation for future library releases, are incompatible.

Objective	RRBC Static	RRBC DLL	C++ Native Static	C++ Native DLL
Reduce code size		X		XX
Reduce load time	X		X	
Eliminate need to recompile with new library releases	X	X		
Eliminate need to rebind with new library releases		X		
Maximize runtime performance			X	X
Avoid dependence on library version installed on target system			X	

C++ SOM and Cross-language SOM Collection Classes

Both C++ SOM and cross-language SOM collection classes are SOM-enabled and therefore support RRBC. Beyond that, the differences are as follows:

- Use the C++ SOM library if you need/wish to:
 - Use C++ template syntax/semantics with classes.
 - Use C++ exception model (throw/catch) to handle exceptions from classes.

- Tailor collection class implementation using pre-defined implementation variants.
- Preserve source compatibility between native C++ and RRBC-enabled applications. Source code is identical between C++ native and C++ SOM classes. However, cross-language SOM classes introduce source incompatibilities in the form of changed class names (eg: IBag to ISBag), operation names, error handling and changed header file suffix (from .h to .hh/.xh) for C++ code.
- Use the cross-language SOM library if you need/wish to:
 - Use the collection classes from C or C++
 - Multiply inherit from a collection class and some SOM class. All classes in a class hierarchy must be SOM classes if any is a SOM class. All cross-language SOM collection classes are derived from SOMObject whereas some C++ SOM collection classes are not.

Coding with Class Libraries under OS/390 OpenEdition Services

If you use the class libraries in applications that run under OS/390 OpenEdition services, the following restrictions apply:

- These class libraries are not safe to use with respect to asynchronous signals. In particular, this means:
 - Do not invoke these classes from a signal handler during asynchronous signal handling.
 - Do not call `longjmp()` or `siglongjmp()` from a signal handler during asynchronous signal handling. If you do, the class library behavior is undefined.

If you don't observe these restrictions, the subsequent behavior of any of these classes is undefined. For simplicity, avoid invoking these classes from any signal handler.

- Do not call `fork()` in user code that has been invoked from class library code (for example, in an override of `IStringTest::test` from the Application Support Class library or in an implementation of `allElementsDo` from the Collection Class library).

Compiling and Binding with the C++ SOM Libraries

The C++ native and C++ SOM class libraries share the same set of header files. Using compile-time switches, you control the inclusion of the correct version of the header files in your application.

Note: If your source code includes the IBM-supplied class library header files, you must use the `SEARCH` compiler option to identify the relevant data sets. Using `SYSLIB` may result in compilation errors.

The IBM-supplied catalog procedure (i.e., `CBCB`, `CBCCB`) binds the C++ native DLL versions of IBM Open Class Library by default. The input definition side-decks are in data set `CBC.SCLBSID`, members `COMPLEX`, `IOSTREAM`, and `ASCCOLL`.

If you want to use the C++ SOM static or DLL class libraries, refer to the directions below.

Note: Your application cannot use multiple copies of an IBM Open Class library. If your application consists of multiple modules (for example, a main module and a DLL) that use the same class library, make sure that all your modules bind dynamically to the class library: otherwise the class library will be linked in multiple times, and there will be multiple copies in use by your application. The use of multiple copies of a class library within a single application is not supported, and can have unexpected results.

Refer to the *OS/390 C/C++ IBM Open Class Library User's Guide* for directions for compiling and binding with the C++ native I/O Stream, Complex Mathematics, Application Support, and Collection Class Libraries, in either the static or DLL forms.

- To use the **DLL C++ SOM** Application Support and Collection Class Libraries, do the following:

1. When you compile, add the following compile options:

```
DEF(__RRBC_LIB__)
SEARCH('CEE.SCEEH.+','CBC.SCLBH.+','SOMMVS.SGOSHH.+','SOMMVS.SGOSH.+',...
```

where SOMMVS is the high level qualifier of the *OS/390 SOMobjects* runtime library.

2. When you bind, include the following definition side-decks on the SYSLIN ddname:

- CBC.SCLBSID(ASCOLSOM)
- CBC.SCLBSID(ISTREAM)
- SOMMVS.SGOSIMP(GOSSOMK)

3. The following libraries must be available at run time:

- CEE.SCEERUN
- CBC.SCLBDLL
- SOMMVS.SGOSLOAD

Note: The C++ SOM DLL (Application Support and Collection Class Libraries) resides in the data set SCLBDLL. The member name is ASCOLSOM.

- To use the **static C++ SOM** Application Support and Collection Class Libraries, do the following:

1. When you compile, add the following compile options:

```
DEF(__RRBC_LIB__)
SEARCH('CEE.SCEEH.+','CBC.SCLBH.+','SOMMVS.SGOSHH.+','SOMMVS.SGOSH.+',...
```

where SOMMVS is the high level qualifier of the *OS/390 SOMobjects* runtime.

2. When you bind:

- Include the following definition side-decks as input:
 - SOMMVS.SGOSIMP(GOSSOMK)
- Remove the following definition side-decks from the binder input:
 - CBC.SCLBSID(ISTREAM/ASCCOLL)
- Include the following static libraries in your autocall library specification, by specifying the following for your SYLIB concatenation.
 - CBC.SCLBOBC
 - CBC.SCLBCPP

3. The following libraries must be available at run time:

- CBC.SCLBDLL
- SOMMVS.SGOSLOAD
- CEE.SCEERUN

The source for the IBM Open Class libraries is available in the CBC.SCLDH.* data sets.

Note: Do not use the class library source data sets (CBC.SCLDH.*) unless you are using your own libraries built from the source in YOURID.ASCSRC or CBC.CCLSRC. If you are, then the class library source data sets must be specified first in the search order.

Migration Notes

The Collection Class Library and the Application Support Class Library DLLs and side-decks were merged in V1R3 of OS/390 C/C++. The combined DLL and side-deck is ASCOLSOM. This single side-deck and DLL should be used for all new applications.

Side-decks with the names APPSUBC (C++ SOM Application Support Class Library entry points only) and COLLRRBC (C++ SOM Collection Class Library entry points only) are supplied for migration purposes only. These side-decks should only be used in circumstances where a name collision exists between your application code and one of the libraries. For example, an application could use the Application Support Class Library and contain a function with the same name as one in the Collection Class Library. In this example, you must bind with the APPSUBC side-deck only to allow the duplicate name to be resolved in the application code. The combined DLL, ASCCOLL, will still be used.

DLLs with the names APPSUBC and COLLRRBC are provided for applications linked with previous releases of OS/390 C/C++. These DLLs are equivalent to ASCOLSOM.

Using the Cross-language SOM Collection Class Library

IDL (Interface Definition Language) for the cross-language SOM collection classes is provided in the CBC.SCLBXL.IDL data set. This IDL can be processed by the SOM compiler to generate bindings for use by the selected language, such as C or C++.

C header files for the cross-language SOM collection classes are provided in the CBC.SCLBXL.H data set.

DTS C++ header files for the cross-language SOM collection classes are provided in the CBC.SCLBXL.HH data set.

The Interface Repository (IR) for the cross-language SOM collection classes is provided in the CBC.SCLBXL.IR data set.

The DLL version of the cross-language SOM Collection Class Library resides as COLLXL in the CBC.SCLBDLL data set. The corresponding definition side-deck is COLLXL and is provided in the CBC.SCLBSID data set. This library requires the C++ native DLLs and side decks IOSTREAM and ASCCOLL.

The static version of the cross-language SOM Collection Class Library resides in the CBC.SCLBOXL data set. This object library requires the C++ native class libraries, CBC.SCLBCPP.

For more information on using IDL to access a class library refer to the *OS/390 SOMobjects Programmer's Guide*.

User's Guide: SOM Cross-language Collection Classes

Chapter 2. Overview of the SOM Cross-language Collection Classes . . .	13
Classes Provided by the Library	13
Benefits of the SOM Cross-language Collection Classes	17
Types of Classes in the SOM Cross-language Collection Classes	17
Flat Collections	18
Restricted Access	24
Auxiliary Classes	24
The Overall Implementation Structure	24
 Chapter 3. Using the Collection Classes	 27
Creating an Operations Class Object	27
Creating Collections	28
Adding, Removing, and Replacing Elements	28
Cursors	30
Iterating over Collections	31
Bounded and Unbounded Collections	33
 Chapter 4. Element Functions and Key-Type Functions	 35
Introduction to Element Functions and Key-Type Functions	35
 Chapter 5. Polymorphic Use of Collections	 37
Introduction to Polymorphism	37
 Chapter 6. Exception Handling	 39
Introduction to Exception Handling	39
Levels of Exception Checking	40
List of Exceptions	40

Chapter 2. Overview of the SOM Cross-language Collection Classes

A SOM collection is an abstract concept, or a SOM class implementing an abstract concept, that allows you to manipulate objects in a group. Collections are used to store and manage elements (or objects). Different collections have different internal structures, and different access methods for storage and retrieval of objects. To be eligible for insertion into any collection, elements must inherit from the `SOMObject` class.

This chapter describes the types of collections provided by the library, introduces the classes that make up the SOM Cross-language Collection Classes, and explains some of the key concepts that are used to describe the SOM Cross-language Collection Classes.

Classes Provided by the Library

This section lists the collections of the Collection Class Library, and provides a verbal description of a potential application for each collection type. The description can be used to aid in the understanding of the characteristics, behavior of each collection, and the overall capabilities of the Collection Classes.

Bag

An example of using a Bag is a program for entering observations on species of plants and animals found in a river. Each time you spot a plant or animal in the river, you enter the name of the species into the collection. If you spot a species twice during an observation period, the species is added twice, because a Bag supports multiple elements. You can locate the name of a species that you have observed, and you can determine the number of observations of that species; however, you cannot sort the collection by species (because a Bag is an unordered collection). To sort the elements of a Bag, you should use a sorted Bag instead.

Deque

An example of using a Deque is a program for managing a lettuce warehouse. Cases of lettuce arriving into the warehouse are registered at one end of the Queue (the “fresh” end) by the receiving department. The shipping department reads the other end of the Queue (the “old” end) to determine which case of lettuce to ship next. However, if an order comes in for very fresh lettuce, which is sold at a premium, the shipping department reads the “fresh” end of the Queue to select the freshest case of lettuce available.

Equality Sequence

An example of using an Equality Sequence is a program that calculates members of the Fibonacci sequence and places them in a collection, with multiple elements of the same value being allowed. For example, the sequence begins with two instances of the value 1. You can search for a given element, for example 8, and find out what element follows it in the sequence. Element equality allows you to accomplish this by using the `locate()` and `setToNext()` functions.

Heap

You can compare using a Heap collection to managing the scrap metal entering a scrapyard. Pieces of scrap are placed in the Heap in an arbitrary location, and an element can be added multiple times (for example, you could have a rear left fender from a particular kind of car). When a customer requests a certain amount of scrap, elements are removed from the Heap in an arbitrary order until the required amount is reached. You cannot search for a specific piece of scrap except by examining each piece of scrap in the Heap and manually comparing it to the piece you are looking for.

Key Bag

An example of using a Key Bag is a program that manages the distribution of combination locks to members of a fitness club. The element key is the number that is printed on the back of each combination lock. Each element also has data members for the club member's name, membership number, and so on. When you join the club, you are given one of the available combination locks, and your name, membership number, and the number on the combination lock are entered into the collection. Because a given number on a combination lock may appear on several locks, the program allows the same lock number to be added to the collection multiple times. When you return a lock because you are leaving the club, the program finds the elements whose key matches your lock's serial number, and deletes the matching element that has your name associated with it.

Key Set

An example of using a Key Set is a program that allocates rooms to patrons checking into a hotel. The room number serves as the element's key, and the patron's name is a data member of the element. When you check in at the front desk, the clerk pulls a room key from the board, and enters that key's number and your name into the collection. When you return the key at check-out time, the record for that key is removed from the collection. You cannot add an element to the collection that is already present, because there is only one key for each room. If you attempt to add an element that is already present, the `add()` function returns 0 to indicate that the element was not added.

Key Sorted Bag

An example of using a Key Sorted Bag is a program that maintains a list of families, sorted by the number of family members in each family. The key is the number of family members. You can add an element whose key is already in the collection (because two families can have the same number of members), and you can generate a list of families sorted by size; however, you cannot locate a family except by its key, because a Key Sorted Bag does not support element equality.

Key Sorted Set

An example of using a Key Sorted Set is a program that keeps track of canceled credit card numbers and the individuals to whom they are issued. Each card number occurs only once, and the collection is sorted by card number. When a merchant enters a customer's card number into a point-of-sale terminal, the collection is checked to see if that card number is listed in the collection of canceled cards. If it is found, the name of the individual is shown, and the merchant is given directions for contacting the credit card company. If the card number is not found, the transaction can proceed because the card is considered to be valid. A list of canceled cards is printed out each month, sorted by card

number, and distributed to all merchants who do not have an automatic point-of-sale terminal installed.

Map

An example of using a Map is a program that translates integer values between the ranges of 0 and 20 to their written equivalents or vice versa (from numeric terms to the written equivalent). Two Maps are created, one with the integer values as keys, one with the written equivalents as keys. You can enter a number, and that number is used as a key to locate the written equivalent. You can enter a written equivalent of a number, and that text is used as a key to locate the value. A given key always matches only one element. You cannot add an element with a key of 1 or “one” if that element is already present in the collection.

Priority Queue

An example of a Priority Queue is a program used to assign priorities to service calls for a heating repair firm. When a customer calls with a problem, a record with that person's name and the seriousness of the situation is placed in a Priority Queue. When a service person becomes available, customers are chosen by the program beginning with those whose situation is most severe. In this example, a serious problem such as a nonfunctioning furnace would be indicated by a low value for the priority, and a minor problem such as a noisy radiator would be indicated by a high value for the priority.

Queue

An example of using a Queue is a program that processes requests for parts at the cash sales desk of a warehouse. A request for a part is added to the Queue when the customer's order is taken, and is removed from the Queue when someone receives the order form for the part. Using a Queue collection in such an application ensures that all orders for parts are processed on a first-come and first-served basis.

Relation

An example of using a Relation is a program that maintains a list of all your relatives, with an individual's relationship, to you, as the key. You can add an aunt, uncle, grandmother, daughter, father-in-law, and so on. You can add an aunt even if an aunt is already in the collection, because you can have several relatives who have the same relationship to you. (For unique relationships such as mother or father, your program would have to check the collection to make sure it did not already contain a family member with that key, before adding the family member.) You can locate a member of the family, but the family members are not in any particular order.

Sequence

An example of a Sequence is a program that maintains a list of the words in a paragraph. The order of the words is obviously important, and you can add or remove words at a given position, but you cannot search for individual words except by iterating through the collection and comparing each word to the word you are searching for. You can add a word that is already present in the sequence, because a given word may be used more than once in a paragraph.

Set

An example of a Set is a program that creates a packing list for a box of free samples to be sent to a warehouse customer. The program searches a database of in-stock merchandise, and selects ten items at random whose price is below a threshold level. Each item is then added to the Set. The Set does not allow an item to be added if it is already present in the collection, ensuring that a customer does not get two samples of a single product. The set is not sorted, and elements of the set cannot be located by key.

Sorted Bag

An example of using a Sorted Bag is a program for entering observations on the types of stones found in a riverbed. Each time you find a stone on the riverbed, you enter the stone's mineral type into the collection. You can enter the same mineral type for several stones, because a sorted Bag supports multiple elements. You can search for stones of a particular mineral type, and you can determine the number of observations of stones of that type. You can also display the contents of the collection, sorted by mineral type, if you want a complete list of observations made to date.

Sorted Map

An example of using a Sorted Map is a program that matches the names of rivers and lakes to their coordinates on a topographical map. The river or lake name is the key. You cannot add a lake or river to the collection if it is already present in the collection. You can display a list of all lakes and rivers, sorted by their names, and you can locate a given lake or river by its key, to determine its coordinates.

Sorted Relation

An example of using a Sorted Relation is a program used by telephone operators to provide directory assistance. The computerized directory is a Sorted Relation whose key is the name of the individual or business associated with a telephone number. When a caller requests the number of a given person or company, the operator enters the name of that person or company to access the phone number. The collection can have multiple identical keys, because two individuals or companies might have the same name. The collection is sorted alphabetically, because once a year it is used as the source material for a printed telephone directory.

Sorted Set

An example of using a sorted set is a program that tests numbers to see if they are prime. Two complementary sorted sets are used, one for prime numbers, and one for nonprime numbers. When you enter a number, the program first looks in the set of nonprime numbers. If the value is found there, the number is nonprime. If the value is not found there, the program looks in the set of prime numbers. If the value is found there, the number is prime. Otherwise the program determines whether the number is prime or nonprime, and places it in the appropriate sorted set. The program can also display a list of prime or nonprime numbers, beginning at the first prime or nonprime following a given value, because the numbers in a sorted set are sorted from smallest to largest.

Stack

An example of using a stack is a program that keeps track of daily tasks that you have begun to work on but that have been interrupted. When you are working on a task and something else comes up that is more urgent, you enter a description of the interrupted task and where you stopped it into your program, and the task is pushed onto the stack. Whenever you complete a present task, you ask the program for the most recently saved task that was interrupted. This task is popped off the stack, and you resume your work where you left off. When you attempt to pop an item off the stack and no item is available, you have completed all your tasks.

Benefits of the SOM Cross-language Collection Classes

In addition to implementing the common abstract data types efficiently and reliably, the SOM Cross-language Collection Classes gives you the following benefits:

- A framework of *properties* to help you decide which abstract data type is appropriate in a given situation
- A choice about how the abstract data type you have chosen is implemented by the SOM Cross-language Collection Classes.

The Collection Class Library lets you choose the appropriate abstract data type for a given situation by providing *collection classes* that are a complete, systematic, and have a consistent combination of basic properties. These properties, which are explained in “Flat Collections” on page 18, help you to select abstract data types that are at the appropriate level of abstraction. In a particular application, for example, you may have the choice between using a Bag and a Key Sorted Set. The properties of these two collections will help you decide which one is more appropriate.

Types of Classes in the SOM Cross-language Collection Classes

The classes that make up the Collection Class Library are divided into two types:

Flat Collections

Flat collections include abstractions such as Sequence, Set, Bag, and Map. Flat collections have no hierarchy of elements or recursive structure, in contrast to trees or graphs for example. All flat collections are derived from a hierarchy of abstract base classes.

See “Flat Collections” on page 18 for more information on flat collections and their properties.

Auxiliary Classes

The *auxiliary classes* include classes for cursors, applicators, comparators, predicates, and operations.

Cursors and applicators give you convenient methods for accessing the elements stored in the collections. See “Cursors” on page 30 for more details on cursor classes. See “Iteration Using Applicators” on page 32 for more details on applicator classes.

Operations are required whenever a collection instance is constructed. They provide the element-type specific information to the collection, for example, they define the ordering relations between the collection's

elements. See Chapter 4, “Element Functions and Key-Type Functions” on page 35 for more details on operations classes.

Flat Collections

Four basic properties are used to differentiate between different flat collections:

Ordering

Whether a *next* or *previous* relationship exists between elements.

Access by key

Whether a part of the element (a *key*) is relevant for accessing an element in the collection. When keys are used, they are compared using relational operators.

Equality for elements

Whether equality is defined for the element.

Uniqueness of entries

Whether any given element or key is *unique*, or whether *multiple* occurrences of the same element or key are allowed.

Figure 2 on page 19 shows the flat collection that results from each combination of properties. For example, “Map” appears in the Unique Unordered column, for the Key Element Equality row. This means that a Map is unordered, each element is unique, keys are defined, and element equality is defined. The figure contains N/A where no flat collection corresponds to the combination of properties. For example, the N/A in the first two rows of the rightmost column indicates that an ordered collection that is sequential (instead of sorted) and offers access by key is not available. This implies that there are no flat collections that have all of the following properties:

- The collection is ordered.
- The collection is sequential.
- The collection allows an element to appear more than once.
- Keys are defined for elements in the collection.

The rationale for not implementing collections with these combinations of properties is that there is no reason to choose them over another collection that is already available. For example, with an ordered collection that is sequential and offers access by key, the key access would only have advantages if the elements are stored in a position depending on their key. Because they are not, there is no flat collection with key access that maintains a sequential order.

		Unordered		Ordered		
				Sorted		Sequential
		Unique	Multiple	Unique	Multiple	Multiple
Key (Key equality must be defined)	Element Equality	Map	Relation	Sorted map	Sorted relation	N/A
	No Element Equality	Key set	Key bag	Key sorted set	Key sorted bag	N/A
No Key	Element Equality	Set	Bag	Sorted set	Sorted bag	Equality sequence
	No Element Equality	N/A	Heap	N/A	N/A	Sequence

Figure 2. Combination of Flat Collection Properties

Ordering of Collection Elements

The elements of a flat collection class can be ordered in three ways:

- *Unordered* collections have elements that are not ordered.
- *Sorted* collections have their elements sorted by an ordering relation defined for the element type. For example, integers can be sorted in ascending order, and strings can be ordered alphabetically. The ordering relation is determined by the instantiations for the collection class. For elements where the ordering relation returns the same position, elements are added in chronological order, that is, in the order as they arrive.
- *Sequential* collections have their ordering determined by an explicit qualifier to the `add()` function, for example, `addAtPosition()`.

A particular element in a sorted collection can be accessed quickly by using the ordering relation to determine its position. Unordered collections can also be implemented to allow fast access to the elements by using, for example, a hash table or a sorted representation. The Collection Class Library provides a fast `locate()` function that uses this structure for unordered and sorted collections. Even though unordered collections are often implemented by sorting the elements, do not assume that all unordered collections are implemented in this way. If your program requires this assumption to be true, use a sorted collection instead.

For each flat collection, the Collection Class Library provides both unordered and sorted abstractions. For example, the Collection Class Library supports both a set and a sorted set. The ordering property is independent of the other properties of flat collections; you have the choice of making a given flat collection unordered or sorted regardless of the choices that you make for the other properties.

Access by Key

A given flat collection can have a *key* defined for its elements. A key is usually a data member of the element, but it can also be calculated from the data members of the element by some arbitrary function. Keys let you:

- Organize the elements in a collection
- Access a particular element in a collection

For collections that have a key defined, an equality relation must be defined for the key. Thus, a collection with a key is said to have *key equality*. In SOM collections where the elements are SOMObjects the key, as the element's data member, must inherit from the class SOMObject. This is not required if the element's key is not a data member but is calculated in some other way.

Equality for Keys and Elements

A flat collection can have an equality relation defined for its elements. The default equality relation is based on the element as a whole, not just on one or more of its data members (for example, the key). For two elements to be equal, all data members of both elements must be equal. The equality relation is needed for functions such as those that locate or remove a given element. A flat collection that has an equality relation has *element equality*.

You can define your own equality relation to behave differently. For example, your equality relation could test only certain data members of two elements to determine element equality. In such cases, element equality may apply to two elements even if the elements are not exactly equal. The equality relation for keys may be different than the equality relation for elements. For more information, refer to the operations functions `Equal()` and `KeyEqual` as described in Chapter 33, "Operations" on page 125.

Conceptually, for example, consider a job control block that has a priority and a job identifier that defines equality for jobs. You could choose to implement a job collection as unordered, with the job ID as key, or as sorted by priority, with the priority as key. The Job class for this job control block could look like this:

```

-----
interface JobId : SOMObject {

#ifdef __SOMIDL__
    implementation {
        unsigned long ivId;
    };
#endif
};
-----

interface Priority : SOMObject {

#ifdef __SOMIDL__
    implementation {
        unsigned long ivId;
    };
#endif
};
-----

interface Job : SOMObject {
    attribute JobId    ivId;
    attribute Priority ivPriority;
#ifdef __SOMIDL__
    implementation {
        releaseorder: _get_ivId, _set_ivId,
                      _get_ivPriority, _set_ivPriority;
    };
#endif
};
-----

// if ivId is the key:
interface JobOps1 : ISOps {

...
Key : override;
};
-----

// if ivId is the key:
SOM_Scope SOMObject* SOMLINK Key(JobOps1 *somSelf, Environment *ev,
                                SOMObject* element)
{
    return ((Job*)element->_get_ivId(ev);
}
-----

// if ivPriority is the key:
interface JobOps2 : ISOps {

...
key : override;
};
-----

// if ivPriority is the key:
SOM_Scope SOMObject* SOMLINK Key(JobOps2 *somSelf, Environment *ev,
                                SOMObject* element)
{
    return ((Job*)element->_get_ivPriority(ev);
}
-----

```

In the first case, you would have fast access through the job ID but not through the priority; in the second case, you would have fast access through the priority but not through the job ID.

All operations that are required to find out whether two elements are equal, what the key of an element is, or what the ordering of elements is within a collection are implemented in a default implementation via an operations class called `ISOps`.

If you let `JobOps1` inherit from `ISOps` and use a `JobOps1` instance within a `Job` initializer method, then `JobId` is defined to be the key. `JobOps2` is used when the Priority should be the key. The overridden `Key` methods of `JobOps1` and `JobOps2` just return the attribute values from the `Job` instance. The ordering relation on the priority key in the second case does not yield a job equality, because two jobs can have equal priorities without being the same.

Functions like `locateElementWithKey()` use the equality relation on keys to locate elements within a collection. A collection that defines key equality may also define element equality. Functions that are based on equality (such as `locate()`) are only provided for collections that define element equality. Collections that define neither key equality nor element equality, such as `Heaps` and `Sequences`, provide no functions for locating elements by their values or testing for containment. Elements can be added and retrieved from such collections by iteration. For sequences, elements can also be added and retrieved by position.

A sorted collection must define either key equality or element equality. A sorted collection that does not have a key defined must have an ordering relation defined for the element type; this relation implicitly defines element equality.

Keys can be used to access a particular element in a collection. The alternative to defining element equality as equality of all data members is to define it as equality of keys only. (In the job control block example on page 20, this means defining job equality as equality of the job ID.) Use this alternative only when you are sure that keys are unique. When you use this alternative, you can locate an element only with the key (using `locateElementWithKey(key)` instead of `locate(element)`). Locating elements by key improves performance, particularly if the complete element is large in comparison to the key alone.

The Collection Class Library provides sorted and unsorted versions of `Maps` and `relations`, for which both key and element equality must be defined. These collections are similar to key set and key bag, except that they define functions based on element equality, namely union and intersection. The `add()` function behaves differently toward `Maps` and `relations` than it does toward key set and key bag.

Uniqueness of Entries

The terms *unique* and *multiple* relate to the key, in collections with a key, and for collections with no key; *unique* and *multiple* relate to the element.

In some flat collections, such as `Map`, key set, and set, no two elements are equal or have equal keys. Such collections are called *unique collections*. Other collections, including relation, key bag, bag, and Heap, can have two equal elements or elements with equal keys. Such collections are called *multiple collections*.

For those multiple collections with key that have element equality (relation and sorted relation), elements are always unique while keys can occur multiple times. In other words, if element equality is defined for a multiple collection with key, element equality is tested before inserting a new element.

A unique collection with no keys and no element equality is not provided because a *containment function* cannot be defined for such a collection. A containment function determines whether a collection contains a given element.

The behavior during element insertion (when one of the `add...` methods is applied to a collection) distinguishes unique and multiple collections. In unique collections, the `add()` function does not add an element that is equal to an element that is already in the collection. In multiple collections, the `add()` function adds elements regardless of whether they are equal to any existing elements or not.

The `add()` function has two general properties:

- All elements that are contained in the collection before an element is added are still contained in the collection after the element is added.
- The element that is added will be contained in the collection after it is added.

Operations that contradict these properties are not valid. You cannot add an element to a Map or sorted map that has the same key as an element that is already contained in the collection. In the case of a Map and sorted map, an exception is thrown.

Note: Both Map and sorted map are unique collections. That is, a request to add an equal element (where not only the key is equal to an already existing key, but the whole element is equal to an existing element) is ignored. The functions `locateOrAddElementWithKey()` and `addOrReplaceElementWithKey()` specify what happens if you try to add an element to a collection that already contains an element with the same key.

Figure 3 shows the result of adding a series of four elements to a Map, a relation, a key set, and a key bag. The first row shows what each collection looks like after the element `<a,1>` has been added to each collection. Each following row shows what the collections look like after the element in the leftmost column is added to each.

The elements are pairs of a character and an integer. The character in the pair is the key. An element equality relation, if defined, holds between two elements if both the character and the integer in each pair are equal.

add	Map or Sorted Map	Relation or Sorted Relation	Key Set or Key Sorted Set	Key Bag or Key Sorted Bag
<code><a,1></code>	<code><a,1></code>	<code><a,1></code>	<code><a,1></code>	<code><a,1></code>
<code><b,1></code>	<code><a,1>, <b,1></code>	<code><a,1>, <b,1></code>	<code><a,1>, <b,1></code>	<code><a,1>, <b,1></code>
<code><a,1></code>	<code><a,1>, <b,1></code>	<code><a,1>, <b,1></code>	<code><a,1>, <b,1></code>	<code><a,1>, <b,1>, <a,1></code>
<code><a,2></code>	exception: Key Already Exists	<code><a,1>, <b,1>, <a,2></code>	<code><a,1>, <b,1></code>	<code><a,1>, <b,1>, <a,1>, <a,2></code>

Figure 3. Behavior of `add` for Unique and Multiple Collections

Restricted Access

Flat collections with restricted access have a restricted set of functions that can be applied to them; that is, only a subset of the functions listed in “Reference: SOM Cross-language Collection Classes - Flat Collections” can be applied. Examples of such flat collections are Stack and priority queue.

You may want to restrict the set of functions for reasons such as:

1. You can simplify the interface to the collection.
2. The normal rules for restricted flat collections apply, so certain assumptions can be made when validating and inspecting the code. A stack, for example, does not allow the removal of any element except the top one.
3. You can create new implementation options.

The Collection Class Library provides the Stack, Deque, and Queue collections, which are based on Sequence, as flat collections with restricted access. These descriptions are alphabetically merged with descriptions for other collections. You can use Table 3 to select the appropriate flat collection with restricted access for a given set of properties.

Table 3. Properties for Collections with Restricted Access

Add	Remove	Sorted (with key)	Unsorted (no key)
According to key	First	Priority Queue	N/A
Last	Last	N/A	Stack
Last	First	N/A	Queue
First or last	First or last	N/A	Deque

Auxiliary Classes

To use the collection classes efficiently, you often need a *cursor* and an *applicator* class. These classes let you iterate over all elements of a collection, and, for example, apply a certain function to all elements or iterate over the collection until you have found a certain element. These classes are described in “Cursors” on page 30 and “Iteration Using Applicators” on page 32.

Read the *OS/390 C/C++ IBM Open Class Library Reference* for a description of Comparators, Predicates, and Operations.

The Overall Implementation Structure

To achieve maximum runtime efficiency and ease of use, the Collection Class Library combines the common features of object-oriented techniques, such as class hierarchies, polymorphism and late binding, with an efficient class structure that uses advanced optimization techniques.

Abstract Classes

The SOM concept of abstract classes differs from that in C++. In C++, the abstract class mechanism supports the notion of a general concept, such as a *shape*, of which only more concrete variants, such as *square* and *circle* can be used. An abstract class not only describes the general concept, it also cannot be instantiated.

Another aspect of the abstract base class is the notion of a *pure virtual function*. Any child of the parent abstract base class must override each pure virtual function (method) in order to use the function.

In SOM, there are no *virtual functions*; however, this C++ concept is similarly valid for SOM. In SOM, class implementors can use one of the following approaches:

- Declare an interface in the parent class to a method that all children must override and redefine. If the method is not overridden, the parent class will raise an exception.
- Declare and define an interface in the parent class to a method that the children can either accept as their base definition or can override and redefine.

The classes in the Collection Class Library are related through a hierarchy of abstract classes shown in Figure 4 on page 26 and Figure 5 on page 26. The leaves of the abstract class hierarchy (that is, those classes that have no derived classes within the abstract class hierarchy tree) define the collection for which concrete implementations are provided. The lines in the figure represent an *is a* relationship from a lower collection to the collection above it. For example, a set is an equality collection, which is a collection. The names of abstract collections start with ISA. There are two separate class hierarchies, namely for flat collections with and without restricted access.

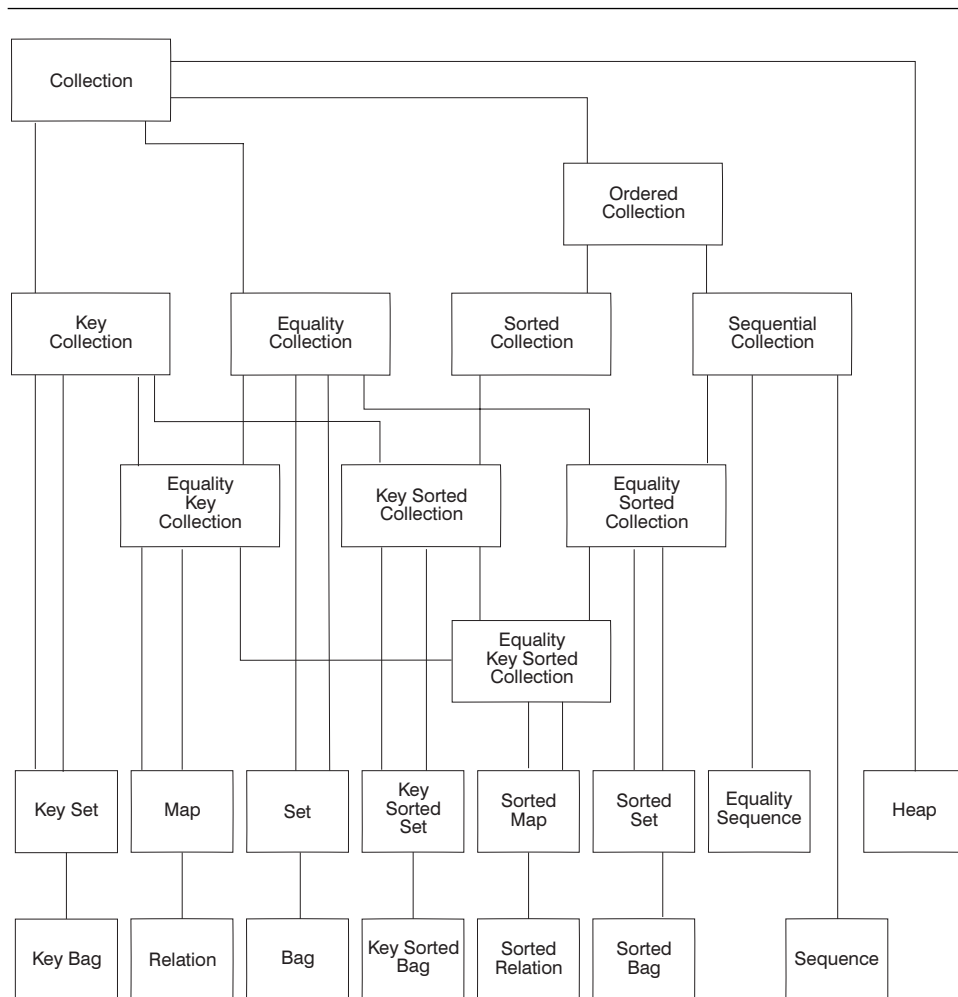


Figure 4. Abstract Hierarchy

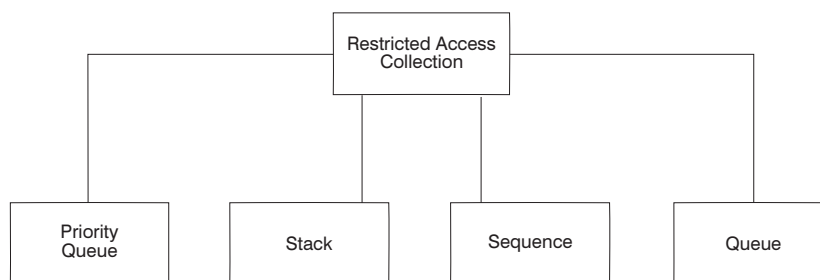


Figure 5. Abstract Hierarchy of Flat Collections with Restricted Access

Chapter 3. Using the Collection Classes

This chapter describes how to use collection classes.

To use a collection class, you normally follow these steps:

1. Select a collection type to be used in your application.
2. Implement a SOM class by subclassing from ISOps.
3. Create an object of above operations class.
4. Create a collection object of the selected collection type, using the operations class object as constructor argument.
5. Apply functions to these objects.
6. Check for exceptions by inspecting the environment structure.

Step 3 is described in more detail in "Creating an Operations Class Object" and step 4 is described in more detail in "Creating Collections" on page 28. All other steps (except step 1) are illustrated by the various samples provided with this documentaion.

Creating an Operations Class Object

Before you create a collection you must first create an operations object. The operations object is required in order to manage elements within a collection. This object provides important element type and key type specific information to the collection implementation.

To define an operations object you must subclass from ISOps, for example,

```
interface AnimalsOps : ISOps {
...
};

...

AnimalsOps          animalsOps ;
...
animalsOps          = (AnimalsOps) AnimalsOpsNew() ;
```

This class contains the superset of all required operations (methods) in a default implementation.

Note!

Each collection requires a certain subset of these operations. See "Required Operations" in the chapters of the individual collections in "Reference: SOM Cross-language Collection Classes - Flat Collections" on page 43. See Chapter 33, "Operations" on page 125 for more information on the default implementation technique for the single operations. You may want to override some or all of the operations required by your selected collection.

Creating Collections

When you construct the collection object you specify an `ISOps` subclass instance as created in “Creating an Operations Class Object” on page 27 within the constructor:

```
observations = ISKeyBagNew_ISKeyBag_withOps(ev, animalsOps) ;
```

After you constructed the collection you must not use the operations any more. The collection is responsible for the destruction of the operations object.

To construct a collection object, you must use the provided initializer methods only; do not use any other SOM defined way to construct a collection object.

For example, for a *bag*, the initializer methods are defined in `sbag.idl`.

Adding, Removing, and Replacing Elements

You can perform three operations to modify a collection:

- Adding elements. Use the `add()` function and its variants.
- Removing elements. Use the `remove()` function and its variants.
- Replacing elements. Use the `replace()` function and its variants.

Adding Elements

The function `add()` places the element identified by its argument into the collection. After an element has been added, all cursors of the collection become undefined. Cursors are used to point to elements of the collection; an undefined cursor is one that might not currently point to a valid element. `add()` behaves differently depending on the properties of the collection:

- In unique collections, an element is not added if it is already contained in the collection.
- In sorted collections, an element is added according to the ordering relation of the collection.
- In sequential collections, an element is added to the end of the collection.

In general, you can copy one collection to another collection that is initially empty by iterating through the elements of the first collection and calling `add()` with each element as an argument.

For sequential collections, elements can be added at a given position using `add` functions other than `add()`, such as `addAtPosition()`, `addAsFirst()`, and `addAsNext()`. Elements after and including the given position are shifted. Positions can be specified by a number, with 1 for the first element, by using the `addAtPosition()` function. Positions can also be specified relative to another element by using the `addAsNext()` or `addAsPrevious()` functions, or relative to the collection as a whole by using the `addAsFirst()` or `addAsLast` functions.

Warning: A potential pitfall exists, depending on how you defined element equality; you may lose object identity when adding an element to a collection.

For example, you defined a Bag of words, with an element equality as case-insensitive string equality, this means that you do not worry about the case but only the sequence of the letters. Therefore, the Bag is 'free to forget about

case' and adding an element will therefore not guarantee that the case is remembered later.

Add the elements `Word`, `word`, and `WORD` to an instance of a `Bag`, in all three of these cases you will receive the return value `true` (element added). If you now iterate through the collection and retrieve the elements one by one you should not expect to receive back `Word`, `word`, and `WORD`; it may be the case that you will receive `WORD` three times.

Removing Elements

In the Collection Classes, you can remove an element that is pointed to by a given cursor by using the `removeAt()` function. All other removal functions operate on the model of first generating a cursor that refers to the desired position and then removing the element to which the cursor refers. There is an important difference between element *values* and element *occurrences*. An element value may, for nonunique collections, occur more than once. The basic `remove()` function always removes only one occurrence of an element.

For collections with key equality or element equality, removal functions remove one or all occurrences of a given key or element. These functions include `remove()`, `removeElementWithKey()`, `removeAllOccurrences()`, and `removeAllElementsWithKey()`. Ordered collections provide functions for removing an element at a given numbered position. Ordered collections also allow you to remove the first or last element of a collection using the `removeFirst()` or `removeLast()` functions.

After an element has been removed, all cursors of the collection become undefined. Therefore, removing all elements with a given property from a collection cannot be done efficiently using cursors. After you have removed one element with the property, the entire collection would have to be searched for the next element with the property. If you want to remove all of the elements in a collection that have a given property, you should use the function `removeAll()` and provide a *predicate* object as its argument. Refer to Chapter 32, "Predicate" on page 123 for an example on how to use *predicate* object.

Replacing Elements

It is possible to modify collections by replacing the value of an element occurrence. Adding and removing elements usually changes the internal structure of the collection. Replacing an element leaves the internal structure unchanged. If an element of a collection is replaced, the cursors in the collection do not become undefined.

For collections that are organized according to element properties, such as an ordering relation, the `replace` function must not change this element property. For key collections, the new key must be equal to the key that is replaced. For nonkey collections with element equality, the new element must be equal to the old element as defined by the element equality relation. The key or element value that must be preserved is called the *positioning property* of the element in the given collection type.

Sequential collections and heaps do not have a positioning property. Element values in sequences and heaps can be changed freely. The `replaceAt()` function checks whether the replacing element has the same positioning property as the

replaced element. (See Chapter 6, “Exception Handling” on page 39 for more details on preconditions.) When you use the `elementAt()` function to replace part of the element value, this check is not performed. If you want to ensure safe replacement (a replacement that does not change the positioning property), use `replaceAt()` rather than `elementAt()`.

Cursors

A *cursor* is a reference to an element in a collection. If the position of the element changes, the cursor is invalidated. This occurs because the cursor refers only to the position of the element and not to the element itself.

A cursor is always associated with a collection. Cursors are implicitly associated with a collection by using the respective collection as a cursor factory. Creating a cursor in a different way than using the collection as cursor factory may result in undefined behaviour. Each collection function that takes a cursor argument has a precondition that the cursor actually belongs to the collection. Simple functions, such as advancing the cursor, are also functions of the cursor itself. For example, given the following definitions within a C application:

```
...
ISSet    myJobSet;
ISCursor myCursor;
...
myCursor = _newCursor(myJobSet, ev);
...
```

the following two lines of code are functionally equivalent:

```
ISCursor_setToNext(myCursor, ev);
ISSet_setToNext(myJobSet, ev, myCursor);
```

You have to use the fully qualified method versions because both interfaces provide methods with the same name.

Cursors and iteration by cursors can be used with any collection. With cursors the Collection Classes provide:

- An iteration scheme that is simpler than using iterators. (See “Iteration Using Applicators” on page 32.)
- The ability to define functions that return cursors. Such functions can give you fast access to an element if it exists, or indicate the non-existence of an element by returning an invalid cursor.

Cursors are only temporarily defined. As soon as elements are added to or removed from the collection, existing cursors become undefined. One of the three following situations occurs:

1. The cursor is invalidated (`isValid()` will return 0).
2. The cursor remains valid and points to an element of the collection; however, it may point to a different element than before.
3. The cursor remains valid but no longer points to an element of the collection.

Do not use an undefined cursor as an argument to a function that requires the cursor to point to an element of the collection.

The abstract class hierarchy defines three methods in order to construct an appropriate cursor for a given collection:

- `newCursor`
- `newElementCursor`
- `newOrderedCursor`

Only above methods should be used to create cursors.

Using Cursors for Locating and Accessing Elements

Cursors provide a basic mechanism for accessing elements of collection classes. For each collection, you can define one or more cursors, and you can use these cursors to access elements. Collection Class functions such as `elementAt()`, `locate()` and `removeAt()` use cursors to access elements.

Several other functions, such as `firstElement()` or `elementWithKey()`, return an element. They can be thought of as first executing a corresponding cursor function, such as `setToFirst()` or `locateElementWithKey()`, and then accessing the element using the cursor.

You must determine if the element exists before trying to access it. If its existence is not known from the context, it must first be checked. To save the extra effort of locating the desired element twice (once for checking whether it exists and then for actually retrieving its reference), use the cursor that is returned by the `locate` function for fast element access:

```
if ( _locateElementWithKey (myCollection, ev, myCursor)) {
    // ...
    myVariable = (MyElement)_elementAt(myCollection, ev, myCursor);
    // ...
}
```

The `elementAt()` function can also be used to replace the value of the referenced element. You must ensure that the value you are changing does not change the positioning property of the element with respect to the given collection. See “Adding, Removing, and Replacing Elements” on page 28 for more details.

Iterating over Collections

Iterating over all or some elements of a collection is a common operation. The Collection Classes give you two methods of iteration:

- Using cursors
- Using the `allElementsDo` function together with applicators

Ordered (including sorted) collections have a well-defined ordering of their elements, while unordered collections have no defined order in which the elements are visited in an iteration; however, each element is visited exactly once.

You cannot add or remove elements from a collection while you are iterating over a collection, or all elements may not be visited once. You cannot use any of the iterations described in this section if you want to remove all of the elements of a collection that have a certain property. Use the function `removeAll()` that takes a

predicate function as argument. See “Removing Elements” on page 29 for details on removing elements.

Iteration Using Cursors

Cursor iteration can be done with a **for** loop. Consider the following example:

```
ISet      myCollection;
MyIntElement currentElement;

// create collection and add elements to collection ...
// ...

ISCursor myCursor = _newCursor(myCollection,ev);
for (ISCursor_setToFirst(myCursor,ev);
     ISCursor_isValid(myCursor,ev);
     ISCursor_setToNext(myCursor,ev))
{
    // ...
    currentElement =
        (MyIntElement) _elementAt(myCollection,ev,myCursor);
    // work with currentElement
    // ...
}
```

myCollection is an instance of ISet. This is the default implementation for a Set. MyIntElement is an interface you derived from SOMObject, you create instances of MyIntElement and add those to myCollection. The for loop iterates over all elements stored within the collection. elementAt returns the current element within a loop iteration.

This code sample does not show any environment tests for possible exceptions.

Note: You should remove multiple elements from a collection using the removeAll() function, with a predicate function as an argument. See “Adding, Removing, and Replacing Elements” on page 28 for further details.

Iteration Using Applicators

Cursor iteration has two possible drawbacks:

- For unordered collections, the explicit notion of an (arbitrary) ordering may be undesirable for stylistic reasons. For example, it could mislead you (or another programmer) into perceiving or exploiting an order where in fact the order does not exist or is not guaranteed.
- Iteration in an arbitrary order might be done more efficiently using something other than cursors. For example, with tree representations, a recursive descent iteration may be faster than the cursor navigation, even though the time for extra function calls must be considered.

The Collection Classes provide the allElementsDo() function that addresses both drawbacks by calling a function that is applied to all elements. The function returns a boolean value which is internally used to indicate continuation or ending of the iteration. For ordered collections, the function is applied in this order. Otherwise the order is unspecified.

The function that is applied in each iteration step can be given by defining the function as a method of a user-defined applicator class:

- **As an object of an applicator class:** Code the function as a member function of an applicator class that you create (for example, *myApplicatorClass*) and let the applicator apply this function to every element, by using `allElementsDo(...,myApplicatorObject)`, where *myApplicatorObject* is an object of *myApplicatorClass*.

Note: You should not add or remove elements while using the applicator.

If you use an object of an applicator class, this class must offer an `applyTo()` function. It also must be derived from the base class `ISApplicator`. The applicator base class is defined in the following way:

```
interface ISApplicator : SOMObject {
    boolean applyTo (in SOMObject element ) ;
    // ...
};
```

Additional arguments that are needed for the iteration can, for example, be passed as arguments to the constructor of the derived applicator class. You must define the function with the given argument and return types.

Bounded and Unbounded Collections

A *bounded* collection limits the number of elements it can contain. There are no bounded collections in the Collection Classes; however, the concept of bounded collections is supported so that you can create your own bounded collection implementations.

When a bounded collection contains the maximum number of elements, the collection is said to be bound and full. This condition can be tested by the function `isFull()`. If elements are added to a full collection, the exception `IFullException` is returned.

You can determine the maximum number of elements in a bounded collection by calling the function `maxNumberOfElements()`. You can only call this function if the collection is bounded. You can determine whether a collection is bounded by calling the function `isBounded()`.

In the current implementation of the Collection Classes, all collections are unbounded. The functions `isBounded()` and `isFull()` always return 0. The function `maxNumberOfElements()` returns the exception `INotBoundedException`.

Chapter 4. Element Functions and Key-Type Functions

This chapter describes the functions that are required by member functions of the Collection Classes to manipulate elements and keys. The following topic is discussed:

- Element operations and key operations defined through operation classes

Note: Using an operations class is the only way to provide element functions and key-type functions for the SOM Cross-language Collection Classes.

Introduction to Element Functions and Key-Type Functions

The member functions of the Collection Class Library call other functions to manipulate elements and keys. These functions are called *element functions* and *key-type functions*, respectively.

Member functions of the Collection Class Library may, for example, need to test the equality relation between elements. When this is required the application programmer must help in doing this job.

The element functions that may be required by a given collection are:

- Assignment
- Equality test
- Ordering relation
- Key access
- Hash function

The key-type functions that may be required by a given collection are:

- Equality test
- Ordering relation
- Hash function

Note: Where both equality test and ordering relation are required element functions (or where both are required key-type functions), the library does not define which of the two is used to determine element or key equality.

The lists above are the superset of all element functions and key-type functions that a Collection Class could ever require. For example, a collection without keys does not require any key-type functions, and a collection without element equality does not require an equality test.

You must subclass from the ISOps interface and override appropriate methods in order to provide above functions.

You can find examples of these functions in the coding examples in the *OS/390 C/C++ IBM Open Class Library Reference*.

The Compare() method must return a value that is less than, equal to, or greater than zero, depending on whether the first argument is less than, equal to, or greater than the second argument. The two arguments for the Compare() method are SOM objects and the application programmer usually issues specific methods against these instances in order to make the decision.

Element and Key-Type Functions

Note: As the default implementation of the provided `Compare()` operation is a pointer comparison, it is advisable to override this operation.

The hash function must return a value that is less than the second argument; the hash function should evenly distribute over the range between zero and the second argument. For equal elements or keys, the hash element must yield equal results. An efficient hash function is very important to the performance of your program.

For `Assign()` a default implementation is defined.

You can also use element operation classes in cases where you want to place elements of one type into more than one collection. For example, suppose you require a collection that is used to store employee records that can be sorted either by name or by salary. You can declare an element class `Person`, and then place references to each `Person` instance into each of two collections. In one collection, the key is the name; in the other, the key is the salary. In your program, you need to define different element and key-type functions for hashing, comparison, and so on.

You then need to define two classes, `SalaryOps` and `NameOps`, which are derived from `ISOps` and which must override appropriate element and key-type methods.

During the construction of the two different sets you specify the two different operations instances.

Chapter 5. Polymorphic Use of Collections

This chapter describes how you can use polymorphism in the Collection Classes.

Introduction to Polymorphism

Polymorphism allows you to take an abstract view of an object or function argument and use any concrete objects or arguments that are derived from this abstract view. The collection properties defined in “Flat Collections” on page 18 define such abstract views. They are represented in the form of the class hierarchy in Figure 4 on page 26.

Polymorphic use of collections differs from polymorphism of the element type.

Polymorphic use of collections means that a function can specify an abstract collection type for its argument, for example `ISACollection`, and then accept any concrete collections given as its actual argument.

Each abstract class is defined by its functions and their behavior. The most abstract view of a collection is a container without any ordering or any specific element or key properties. Elements can be added to a collection, and a collection can be iterated over. A polymorphic function on collections might be to print all elements, such a function is given as an example on page 37.

Example

```
interface JobPrinter : SOMObject {
    void print (in ISACollection jobs) ;

    // ...

};

interface Job : SOMObject {

    // ...

};

// use JobId as the key within a Job
interface JobId : SOMObject {

    // ...

};

... put C++ implementation stubs for print here...

ISKeySet running;
JobPrinter printer = JobPrinterNew(0,0);

// create Job instances and add to running Key Set...

_print(printer,ev,running);
```

Polymorphic Use of Collections

The `print` method of `JobPrinter` is coded to process objects of type `ISACollection`. `ISKeySet` is derived from `ISACollection` and can be used as parameter to the `print` method.

Collections whose elements define equality or key equality provide, in addition to the common collection functions, functions for retrieving element occurrences by a given element or key value. Ordered collections provide the notion of a well-defined ordering of the element occurrences, either by an element ordering relation or by explicit positioning of elements within a sequence. They define operations for positional element access. Sorted collections provide no further functions, but define a more specific behavior, namely that the elements or their keys are sorted.

These properties are combined through multiple inheritance; the abstract collection class `ISAEqualitySortedCollection`, for example, combines the abstract concepts of element equality and of being sorted, which implies being ordered. If a polymorphic function uses this class as its argument type, the arguments will be sorted, and the function can use functions like `contains()` that are only defined for collections with element equality.

Chapter 6. Exception Handling

This chapter describes the exception-handling facilities provided by member functions of the Collection Class Library. The following topics are discussed in this chapter.

- Introduction to exception handling
- Preconditions and defined behavior
- Levels of exception checking
- List of exceptions

Introduction to Exception Handling

An exception is a user, logic, or system error that is detected by a function that does not itself deal with the error, but passes the error to a handling function. Exceptions can result from two major sources:

- The violation of a precondition
- The occurrence of an internal system failure or system restriction

In this chapter, two kinds of functions are discussed. A *called* function is a Collection Class function that may throw an exception. A *calling* function is a function that calls a Collection Class function. The *calling* function may be a Collection Class function or a function you have defined.

It is the responsibility of the collection class user to check the SOM environment structure after every collection method call for possible exceptions. In case the `_major` field of the current environment has a value other than `USER_EXCEPTION` the SOM function `somExceptionId()` must be called to inspect the exception code. It is also the user's responsibility to free the exception space by calling `somExceptionFree()`.

Exceptions Caused by Violated Preconditions

A *precondition* of a called function is a condition that the function requires to be true when it is called. The calling function must assure that this condition holds. The called function implementation may assume that the condition holds without further checking it. If a precondition does not hold, the called function's behavior is undefined.

If you want to make your programs more robust and to locate errors in the test phase, the functions your program calls should check to ensure that their preconditions hold. The Collection Class Library enables this checking through a specific collection initializer method. Because this checking often requires significant overhead, it is turned off by default. You need only use it while you are testing the system and verifying that preconditions are always met.

A call to a function that violates the function's preconditions has two possible results:

- If the called function checks its preconditions, the function will provide an exception within the current environment structure.
- If the function does not check its preconditions, the behavior of the function is undefined.

Exceptions Caused by System Failures and Restrictions

System failures and restrictions are different from precondition violations. You cannot usually anticipate them, and you have no opportunity to verify that such situations, for example storage overflow, will not occur.

Levels of Exception Checking

Some preconditions are more difficult to check than others. Consider the following possible preconditions:

1. A cursor for a linked collection implementation still points to an element of a given collection.
2. A collection is not empty.

In the production version of a program, it may be less efficient to check the first precondition than the second.

List of Exceptions

The Collection Class Library defines the following exceptions:

Notes:

1. All exceptions contain the prefix `::ISOM`; for example, `::ISOMICursorInvalidException`
2. The exception string is returned by `somExceptionId()` and there is no value returned from `somExceptionValue()`

ISApplicatorOverrideException

The current applicator argument does not have a valid `applyTo()` method. You must subclass from `ISApplicator` and override the `applyTo()` method. Then you must create an instance of your applicator class which must be specified in the current collection method call.

ISComparatorOverrideException

The current comparator argument does not have a valid `compare()` method. You must subclass from `ISComparator` and override the `compare()` method. Then you must create an instance of your comparator class which must be specified in the current collection method call.

ICursorInvalidException

Two cursor properties may lead to the `ICursorInvalidException`:

- Every time a cursor is created, you must specify the collection that it belongs to. If a function takes a cursor as an argument (such as `add()`, `setToFirst()`, and `locate()`), the function can only be applied to the collection that the cursor belongs to. If the function is applied to another collection, the result is an `ICursorInvalidException`.
- If a function takes a cursor as an input argument (such as `elementAt()`, `removeAt()`, and `replaceAt()`), the cursor must be *valid*. A cursor is valid if it actually refers to some element contained in the collection. You can use the `isValid()` function to determine if a cursor is valid.

ICollectionEmptyException

Occurs when a function tries to access an element of an empty collection. Functions that might cause this exception include `firstElement()` and `removeFirstElement()`.

ICollectionFullException

Occurs when a function tries to add an element to a bounded collection that is already full. Functions that might cause this exception include `add()` and `addAsFirst()`.

ICollectionIdenticalCollectionException

Occurs when the function `addAllFrom()` is called with the source collection being the same as the target collection.

ICollectionInvalidObjectException

A method called for the current collection failed. Most likely the collection initializer failed. Check if a valid initializer was used. Make sure that the no exception occurred during collection initializing by checking the SOM environment structure after the initializer call.

ICollectionInvalidReplacementException

Occurs when, during a `replaceAt()` function, the replacing element has different positioning properties (see “Replacing Elements” on page 29) than the positioning properties of the element to be replaced.

ICollectionKeyAlreadyExistsException

Occurs when a function attempts to add an element to a map or sorted map that already has a different element with the same key. Functions that might cause this exception include `add` and `addAllFrom()`.

ICollectionNotBoundedException

Occurs when the function `maxNumberOfElements()` is applied to a collection that is not bounded.

ICollectionNotContainsKeyException

Occurs when the function `elementWithKey()` is applied to a collection that does not contain an element with the specified key.

ICollectionOpsInUseException

The specified operations object within the current initializer call is not valid for this collection. Most likely it is already used for another collection. Each collection must have its own unique operations object instance. You must not use the same operations instance for more than one collection.

ICollectionOutOfCollectionMemoryException

Occurs when a function cannot obtain the space that it requires. This exception is not the result of a precondition violation. Functions that add an element to a collection, including `add()` and `addAsFirst()`, can cause this exception.

IPositionInvalidException

Occurs when a function specifies a position that is not valid in a collection. The functions that might cause this exception include `elementAtPosition()`, `removeAtPosition()`, and `setToPosition()`.

IPredicateOverrideException

The current predicate argument does not have a valid `evaluateFor()` method. You must subclass from `ISPredicate` and override the `evaluateFor()` method. Then you must create an instance of your predicate class which must be specified in the current collection method call.

IRemoteCollectionException

Is not allowed to specify a remote collection as one of the arguments in the current method call.

INoSOMObjectException

A specified argument which is supposed to be a SOM object is invalid. It is either a NIL pointer or the internally called `somIsObj()` call failed.

IUserApplicatorException

An exception occurred within the user provided code of the overridden `ISApplicator` method `applyTo()`.

IUserComparatorException

An exception occurred within the user provided code of the overridden `ISComparator` method `compare()`.

IUserOpsException

An exception occurred within the user provided code of an overridden `ISOps` method.

IUserPredicateException

An exception occurred within the user provided code of the overridden `ISPredicate` method `evaluateFor()`.

Reference: SOM Cross-language Collection Classes - Flat Collections

Chapter 7. Introduction to Flat Collections	45
Terms Used	45
Chapter 8. Flat Collection Member Functions	47
Chapter 9. Bag	73
Chapter 10. Deque	75
Chapter 11. Equality Sequence	77
Chapter 12. Heap	79
Chapter 13. Key Bag	81
Chapter 14. Key Set	83
Chapter 15. Key Sorted Bag	85
Chapter 16. Key Sorted Set	87
Chapter 17. Map	89
Chapter 18. Priority Queue	91
Chapter 19. Queue	93
Chapter 20. Relation	95
Chapter 21. Sequence	97
Chapter 22. Set	99
Chapter 23. Sorted Bag	101
Chapter 24. Sorted Map	103
Chapter 25. Sorted Relation	105
Chapter 26. Sorted Set	107
Chapter 27. Stack	109

Chapter 7. Introduction to Flat Collections

This chapter defines some of the terms used in describing the Collection Class Library classes, details the format of chapters that describe individual collections, and describes some types defined by the Collection Class Library.

Terms Used

INTERF_BASE_NAME

For constructor declarations, this term is used in place of one of several initializer methods as specified within the appropriate IDL. For example, a constructor `INTERF_BASE_NAME(...)` for a Bag, can really be `_ISBag_withOpsnNumber(...)` for a C application programmer. You can find the actual names in the related IDL files, for example, for a *bag*, in file `sbag.idl`. Depending on the chosen arguments other initializer methods are possible

INTERF_NAME For member function declarations, this term is used in place of the interface name arguments. For example, if you want to use:

```
void addIntersection ( in INTERF_NAME collection1,
                      in INTERF_NAME collection2 );
```

for a Bag, substitute `ISABag` for `INTERF_NAME`.

INTERF_CORE Denotes the core part of an interface name; for example, Bag is the core part in `ISBag` and Map is the core part in `ISMap`

equal element Refers to equality of elements as defined by the equality operation or ordering relation provided through the class object inherited from `ISOps` when the collection is constructed. Where both equality operation and ordering relation are provided, the Collection Class Library may use either to determine element equality.

given ... Refers to an argument of the described function, such as given element, given key, or given collection.

iteration order The order in which elements are visited in `allElementsDo()` and `setToNext()` or `setToPrevious()`.

In ordered collections, the element at position 1 will be visited first, then the element at position 2, and so on. Sorted collections, in particular, are visited following the ordering relation provided for the element type.

In collections that are not ordered, the elements are visited in an arbitrary order. Each element is visited exactly once.

positioning property

The property of an element that is used to position the element in a collection. For key collections, the positioning property is key equality. For non-sequential collections with element equality, the positioning property is element equality. Other collections have no positioning property.

same key	Refers to equality of keys as defined by the equality operation or ordering relation provided for the key type. Where both equality operation and ordering relation are provided, the Collection Class Library may use either to determine key equality.
this collection	The collection to which a function is applied. Contrast with the <i>given</i> collection, which is an argument supplied to a function. <i>The collection</i> is synonymous with <i>this collection</i> .
undefined cursor	A cursor that may or may not be valid; there is no way to know whether the cursor is valid or not. An undefined cursor, even if it remains valid, may refer to a different element than before, or even to no element of the collection. Do not use cursors, once they become undefined, in functions that require the cursor to point to an element of the collection.

Notes:

1. None of the described interfaces are thread-safe.
 2. SOM related arguments like the Environment pointer or additional arguments used within initializer methods are not shown.
 3. C and C++ programmers must include `ssglobal` with the language dependent suffix before any further collection specific usage bindings are included. The IDL filestem denotes the filename for the usage binding include file of a specific collection.
 4. Additional to the listed exceptions any method may return the exception `::ISOMIInvalidObjectException`. In case a method requires the specification of a SOM Object argument the additional exception `::ISOMINoSOMObjectException` could be returned. `::ISOMIRemoteCollectionException` may be returned if a method allows the specification of another collection as method argument.
 5. Depending on the overwritten methods of the operation's interface `ISOps`, several methods may raise the additional exception `::ISOMIUserOpsException`.
 6. Whenever a method name collides with a method used within an interface not described in this document it is dependent on the used language bindings if fully qualified method names must be used. The cross-language SOM interfaces do not override the following:
 - `somDefaultCopyInit`,
 - `somDefaultAssign`
 - `somDefaultConstAssign`
 - `somDefaultConstCopyInit`
- If the user uses the above methods, unpredictable behavior might occur.
7. If you are a C++ user, and a method requires the specification of an ISACollection parameter, for example **`addAllFrom()`**, you must first cast the parameter to `void *`.

Chapter 8. Flat Collection Member Functions

Initializer Methods

INTERF_BASE_NAME_withCNumber

(inout somInitCtrl ctrl, in long number, in ICheck check);

INTERF_BASE_NAME_withCOps

(inout somInitCtrl ctrl, in ISOPs ops, in ICheck check);

INTERF_BASE_NAME_withCOpsnNumber

(inout somInitCtrl ctrl, in ISOPs ops, in long number, in ICheck check)

INTERF_BASE_NAME_withNumber

(inout somInitCtrl ctrl, in long number);

INTERF_BASE_NAME_withOps

(inout somInitCtrl ctrl, in ISOPs ops);

INTERF_BASE_NAME_withOpsnNumber

(inout somInitCtrl ctrl, in ISOPs ops, in long number);

Constructs a collection, with *numberOfElements* is the estimated maximum number of elements contained in the collection. The collection is unbounded and is initially empty. If the estimated maximum is exceeded, the collection is automatically enlarged.

INTERF_BASE_NAME_withINTERF_CORE

(inout somInitCtrl ctrl, in INTERF_BASE_NAME collection);

Constructs a collection and copies all elements from the given collection into the collection as described for `addAllFrom()`, in “`addAllFrom`” on page 48.

Notes:

1. Most collections require the specification of the *ops* argument. It defines ordering relations for elements as well as for keys which are required when storing elements within collections.
2. The check parameter can be set to *eCheck*, which chooses an internal implementation variant with extensive precondition checkings.

Exceptions

- `IOpsInUseException`

Exceptions

- `IOutOfCollectionMemoryException`

Destructor

Removes all elements from the collection. Refer to the *OS/390 V1R3.0 SOMobjects User's Guide* for information on how to delete a `SOMObject`.

Side Effects: All cursors of the collection become undefined.

add

```
boolean add ( in SOMObject element ) ;
```

```
boolean addWithCursor ( in SOMObject element,  
                        in ISCursor cursor ) ;
```

If the collection is unique (with respect to elements or keys) and the element or key is already contained in the collection, the cursor is set to the existing element in the collection without adding the element. Otherwise, it adds the element to the collection and sets the cursor to the added element. In sequential collections, the given element is added as the last element. In sorted collections, the element is added at a position determined by the element or key value. Adding an element internally copies a pointer to the element into the collection. See “contains” on page 55 for the definition of element or key containment. See “Adding Elements” on page 28 for potential problems associated with an add().

Preconditions

- The cursor must belong to the collection.
- If the collection is bounded and unique, the element or key must exist or `(numberOfElements() < maxNumberOfElements())`.
- If the collection is bounded and nonunique, `(numberOfElements() < maxNumberOfElements())`.
- If the collection is a map or a sorted map and contains an element with the same key as the given element, this element must be equal to the given element.

Side Effects: If an element was added, all cursors of this collection, except the given cursor, become undefined.

Return Value: Returns 1 if the element was added.

Exceptions

- `IOutOfCollectionMemoryException`
- `ICursorInvalidException`
- `IFullException`, if the collection is bounded
- `IKeyAlreadyExistsException`, if the collection is a map or a sorted map

addAllFrom

```
void addAllFrom ( in ISACollection collection ) ;
```

Adds (copies) all elements of the given collection to the collection. The elements are added in the iteration order of the given collection. The elements are added according to the definition of add for this collection. The given collection is not changed.

Preconditions: Because the elements are added one by one, the following preconditions are tested for each individual add operation:

- If the collection is bounded and unique, the element or key must exist or `(numberOfElements() < maxNumberOfElements())`.
- If the collection is bounded and nonunique, `(numberOfElements() < maxNumberOfElements())`.

- If the collection is a map or a sorted map, and contains an element with the same key as the given element, this element must be equal to the given element.

Side Effects: If any elements were added, all cursors of this collection become undefined.

Exceptions

- `IOutOfCollectionMemoryException`
- `IIIdenticalCollectionException`
- `IFullException`, if the collection is bounded
- `IKeyAlreadyExistsException`, if the collection is a map or a sorted map

addAsFirst

```
void addAsFirst ( in SOMObject element ) ;
```

```
void addAsFirstWithCursor ( in SOMObject element  
    in ISCursor cursor ) ;
```

Adds the element to the collection as the first element in sequential order and sets the cursor to the added element.

Preconditions

- The cursor must belong to the collection.
- If the collection is bounded, `(numberOfElements() < maxNumberOfElements())`.

Side Effects: All cursors of this collection, except the given cursor, become undefined.

Exceptions

- `ICursorInvalidException`
- `IOutOfCollectionMemoryException`
- `IFullException`, if the collection is bounded

addAsLast

```
void addAsLast ( in SOMObject element ) ;
```

```
void addAsLastWithCursor ( in SOMObject element,  
    in ISCursor cursor ) ;
```

Adds the element to the collection as the last element in sequential order and sets the cursor to the added element.

Preconditions

- The cursor must belong to the collection.
- If the collection is bounded, `(numberOfElements() < maxNumberOfElements())`.

Side Effects: All cursors of this collection, except the given cursor, become undefined.

Exceptions

- `ICursorInvalidException`
- `IOutOfCollectionMemoryException`

- `IFullException`, if the collection is bounded

addAsNext

```
void addAsNext ( in SOMObject element,  
                in ISCursor cursor ) ;
```

Adds the element to the collection as the element following the element pointed to by the cursor and sets the cursor to the added element.

Preconditions

- The cursor must belong to the collection and must point to an element of the collection.
- If the collection is bounded, `(numberOfElements() < maxNumberOfElements())`.

Side Effects: All cursors of this collection, except the given cursor, become undefined.

Exceptions

- `IOutOfCollectionMemoryException`
- `ICursorInvalidException`
- `IFullException`, if the collection is bounded

addAsPrevious

```
void addAsPrevious ( in SOMObject element,  
                   in ISCursor cursor ) ;
```

Adds the element to the collection as the element preceding the element pointed to by the cursor and sets the cursor to the added element.

Preconditions

- The cursor must belong to the collection and must point to an element of the collection.
- If the collection is bounded, `(numberOfElements() < maxNumberOfElements())`.

Side Effects: All cursors of this collection, except the given cursor, become undefined.

Exceptions

- `IOutOfCollectionMemoryException`
- `ICursorInvalidException`
- `IFullException`, if the collection is bounded

addAtPosition

```
void addAtPosition ( in unsigned long position,  
                   in SOMObject element ) ;
```

```
void addAtPositionWithCursor ( in unsigned long position,  
                              in SOMObject element, in ISCursor cursor ) ;
```

Adds the element at the given position to the collection, and sets the cursor to the added element. If an element exists at the given position, the new element is added as the element preceding the existing element.

Preconditions

- The cursor must belong to the collection.
- $(1 \leq position \leq numberOfElements + 1)$.
- If the collection is bounded, $(numberOfElements() < maxNumberOfElements())$.

Side Effects: All cursors of this collection, except the given cursor, become undefined.

Exceptions

- `IOutOfCollectionMemoryException`
- `ICursorInvalidException`
- `IPositionInvalidException`
- `IFullException`, if the collection is bounded

addDifference

```
void addDifference ( in INTERF_NAME collection1,
                   in INTERF_NAME collection2 ) ;
```

Creates the difference between the two given collections, and adds this difference to the collection. The contents of the added elements, not the pointers to those elements, are copied.

Note: For a definition of the difference between two collections, see “differenceWith” on page 56.

Preconditions: Because the elements are added one by one, the following preconditions are tested for each individual addition.

- If the collection is bounded and unique, the element or key must exist or $(numberOfElements() < maxNumberOfElements())$.
- If the collection is bounded and nonunique, $(numberOfElements() < maxNumberOfElements())$.
- If the collection is a map or a sorted map and contains an element with the same key as the given element, this element must be equal to the given element.

Side Effects: If any elements were added, all cursors of this collection become undefined.

Exceptions

- `IOutOfCollectionMemoryException`
- `IFullException`, if the collection is bounded
- `IKeyAlreadyExistsException`, if the collection is a map or a sorted map

addIntersection

```
void addIntersection ( in INTERF_NAME collection1,
                      in INTERF_NAME collection2 ) ;
```

Creates the intersection of the two given collections, and adds this intersection to the collection.

Note: For a definition of the intersection of two collections, see “intersectionWith” on page 58.

Preconditions: Because the elements are added one by one, the following preconditions are tested for each individual addition.

- If the collection is bounded and unique, the element or key must exist or `(numberOfElements() < maxNumberOfElements())`.
- If the collection is bounded and nonunique, `(numberOfElements() < maxNumberOfElements())`.
- If the collection is a map or a sorted map and contains an element with the same key as the given element, this element must be equal to the given element.

Side Effects: If any elements were added, all cursors of this collection become undefined.

Exceptions

- `IOutOfCollectionMemoryException`
- `IFullException`, if the collection is bounded
- `IKeyAlreadyExistsException`, if the collection is a map or a sorted map

addOrReplaceElementWithKey

```
boolean addOrReplaceElementWithKey (  
    in SOMObject element );
```

```
boolean addOrReplaceElementWithKeyWithCursor (  
    in SOMObject element, in ISCursor cursor ) ;
```

If an element is contained in the collection where the key is equal to the key of the given element, the cursor is set to this element in the collection and replaces it with the given element. Otherwise, it adds the given element to the collection, and sets the cursor to the added element. If the given element is added, the contents of the element, not a pointer to it, is added.

Preconditions

- The cursor must belong to the collection.
- If the collection is bounded, an element with the given key must be contained in the collection, or `(numberOfElements() < maxNumberOfElements())`.

Side Effects: If the element was added, all cursors of this collection, except the given cursor, become undefined.

Return Value: Returns 1 if the element was added. Returns 0 if the element was replaced.

Exceptions

- `IOutOfMemoryException`
- `ICursorInvalidException`
- `IFullException`, if the collection is bounded

addUnion

```
void addUnion ( in INTERF_NAME collection1,
               in INTERF_NAME collection2 ) ;
```

Creates the union of the two given collections, and adds this union to the collection.

Note: For a definition of the union of two collections, see “unionWith” on page 71.

Preconditions: Because the elements are added one by one, the following preconditions are tested for each individual addition.

- If the collection is bounded and unique, the element or key must exist or (numberOfElements() < maxNumberOfElements()).
- If the collection is bounded and nonunique, (numberOfElements() < maxNumberOfElements()).
- If the collection is a map or a sorted map and contains an element with the same key as the given element, this element must be equal to the given element.

Side Effects: If any elements were added, all cursors of this collection become undefined.

Exceptions

- IOutOfCollectionMemoryException
- IFullException, if the collection is bounded
- IKeyAlreadyExistsException, if the collection is a map or a sorted map

allElementsDo

```
boolean allElementsDo ( in ISApplicator applicator ) ;
```

Calls the applyTo() method of the given applicator for all elements of the collection until the applyTo() function returns 0. The elements are visited in iteration order. (For further details, see “Iteration Using Applicators.”)

Notes:

1. The applyTo() function must not remove elements from or add elements to the collection. If you want to remove elements, you can use the removeAll() function with a property argument. For further information, see “removeAllWithPredicate” on page 66.
2. The applyTo() method must not manipulate the element in the collection in a way that changes the positioning property of the element.

Return Value: Returns 1 if the applyTo() method returns 1 for every element it is applied to.

Exceptions

- IApplicatorOverrideException
- IUserApplicatorException

anyElement

SOMObject **anyElement** () ;

Returns a SOMObject pointer to an arbitrary element of the collection.

Precondition: The collection must not be empty.

Exceptions

- IEmptyException

assign

void **assign** (in INTERF_NAME *collection*) ;

Copies the given collection to the collection. Removes all elements from the collection and adds the elements from the given collection as described for “addAllFrom” on page 48.

Preconditions

- If the collection is bounded, numberOfElements() of the given collection must be less than maxNumberOfElements() of this collection.

Side Effects: All cursors of this collection become undefined.

Exceptions

- IOutOfCollectionMemoryException
- IFullException, if the collection is bounded

compare

long **compare** (in INTERF_NAME *collection*,
in ISComparator *comparator*) ;

Compares the collection with the given collection. Comparison yields <0 if the collection is less than the given collection, 0 if the collection is equal to the given collection, and >0 if the collection is greater than the given collection. Comparison is defined by the first pair of corresponding elements, in both collections, that are not equal. If such a pair exists, the collection with the greater element is the greater one. Otherwise, the collection with more elements is the greater one.

Notes:

1. The compare method of the user's ISComparator subclass object must return a result according to the following rules:

>0	if (element1 > element2)
0	if (element1 == element2)
<0	if (element1 < element2)

Return Value: Returns the result of the collection comparison.

Exceptions

- IComparatorOverrideException
- IUserComparatorException

contains

boolean **contains** (in SOMObject *element*) ;

Returns 1 if the collection contains an element equal to the given element.

containsAllFrom

boolean **containsAllFrom** (in ISACollection *collection*) ;

Returns 1 if all the elements of the given collection are contained in the collection. The definition of containment is described in “contains.”

containsAllKeysFrom

boolean **containsAllKeysFrom** (in ISACollection *collection*) ;

Returns 1 if all of the keys of the given collection are contained in the collection.

containsElementWithKey

boolean **containsElementWithKey** (in SOMObject *key*) ;

Returns 1 if the collection contains an element with the same key as the given key.

copy

void **copy** (in ISACollection *collection*) ;

Copies the given collection to this collection, essentially `copy()` removes all elements from this collection, and adds the elements from the given collection. For information on how adding is done, see “addAllFrom” on page 48.

Note: The given collection may be of a concrete type other than the collection itself. In this case, copying implicitly performs a conversion. If, for example, the given collection is a bag and the collection itself is a set, elements with multiple occurrences in the copied bag will only occur once in the resulting set.

Preconditions: Because the elements are copied one by one, the following preconditions are tested for each individual copy operation:

- If the collection is bounded and unique, the element or key must exist or `(numberOfElements() < maxNumberOfElements())`.
- If the collection is bounded and nonunique, `(numberOfElements() < maxNumberOfElements())`.
- If the collection is a map or a sorted map and contains an element with the same key as the given element, this element must be equal to the given element.

Side Effects: All cursors of this collection become undefined.

Exceptions

- `IOutOfCollectionMemoryException`
- `IFullException`, if the collection is bounded
- `IKeyAlreadyExistsException`, if the collection has unique keys. This exception may be thrown, for example, when copying a bag into a map.

deque

void **Deque** () ;

void **DequeWithElement** (in SOMObject* *element*) ;

Copies the first element of the collection to the given element, and removes it from the collection.

Precondition: The collection must not be empty.

Side Effects: All cursors of this collection become undefined.

Exceptions

- IEmptyException

differenceWith

void **differenceWith** (in INTERF_NAME *collection*) ;

Makes the collection the difference between the collection and the given collection. The *difference* of A and B (A minus B) is the set of elements that are contained in A but not in B. (and conversely as well).

The following rule applies for bags with duplicate elements:

If bag P contains the element X m times and bag Q contains the element X n times, the *difference* of P and Q contains the element X $m-n$ times if $m > n$, and zero times if $m \leq n$.

Side Effects: If any elements were removed, all cursors of this collection become undefined.

elementAt

SOMObject **elementAt** (in ISCursor *cursor*) ;

Returns a SOMObject pointer to the element pointed to by the given cursor.

Note: Do not manipulate the element or the key of the element in the collection in a way that changes the positioning property of the element.

Precondition: The cursor must belong to the collection and must point to an element of the collection.

Exceptions

- ICursorInvalidException

elementAtPosition

SOMObject **elementAtPosition** (
 unsigned long *position*) ;

Returns the element at the given position in the collection.

Position 1 specifies the first element.

Position must be a valid position in the collection; that is, $(1 \leq \textit{position} \leq \text{numberOfElements}())$.

Precondition: $(1 \leq \textit{position} \leq \text{numberOfElements}())$.

Exceptions

- `IPositionInvalidException`

elementWithKey

`SOMObject elementWithKey (in SOMObject key) ;`

Returns the element specified by the key.

Notes:

Do not manipulate the element in the collection in a way that changes the positioning property of the element.

1. If there are several elements with the given key, an arbitrary one is returned.

Precondition: The given key is contained in the collection.

Exceptions

- `INotContainsKeyException`

enqueue

`void enqueue (in SOMObject element) ;`

`void enqueueWithCursor (in SOMObject element,
in ISCursor cursor) ;`

Adds the element to the collection, and sets the cursor to the added element. For ordinary queues, the given element is added as the last element. For priority queues, the element is added at a position determined by the ordering relation provided for the element or key type.

Preconditions

- The cursor must belong to the collection.
- If the collection is bounded, $(\text{numberOfElements}() < \text{maxNumberOfElements}())$.

Side Effects: All cursors of this collection except the given cursor become undefined.

Exceptions

- `IOutOfCollectionMemoryException`
- `ICursorInvalidException`
- `IFullException`, if the collection is bounded

equal

boolean **equal** (in INTERF_NAME *collection*) ;

Returns 1 if the given collection is equal to the collection. Two collections are equal if the number of elements in each collection is the same, and if the condition for the collection is in conformance with the following list:

Type of Collection	Condition
Unique Elements	If the collections have unique elements, any element that occurs in one collection must occur in the other collection.
Non-Unique Elements	If an element has n occurrences in one collection, it must have exactly n occurrences in the other collection.
Sequential	The ordering of the elements is the same for both collections.

firstElement

SOMObject **firstElement** () ;

Returns the first element of the collection.

Precondition: The collection must not be empty.

Exceptions

- IEmptyException

intersectionWith

void **intersectionWith** (in INTERF_NAME *collection*) ;

Makes the collection the intersection of the collection and the given collection. The *intersection* of A and B is the set of elements that is contained in both A and B.

The following rule applies for bags with duplicate elements: If bag P contains the element X m times and bag Q contains the element X n times, the *intersection* of P and Q contains the element X $\text{MIN}(m,n)$ times.

Side Effects: If any elements were removed, all cursors of this collection become undefined.

isBounded

boolean **isBounded** () ;

Returns 1 if the collection is bounded.

isEmpty

boolean **isEmpty** () ;

Returns 1 if the collection is empty.

isFirst

boolean **isFirst** (in ISCursor *cursor*) ;

Returns 1 if the given cursor points to the first element of the collection.

Preconditions: The cursor must belong to the collection and must point to an element of the collection.

Exceptions

- ICursorInvalidException

isFull

boolean **isFull** () ;

Returns 1 if the collection is bounded and contains the maximum number of elements; that is, if (numberOfElements() == maxNumberOfElements()).

isLast

boolean **isLast** (in ISCursor *cursor*) ;

Returns 1 if the given cursor points to the last element of the collection.

Preconditions: The cursor must belong to the collection and must point to an element of the collection.

Exceptions

- ICursorInvalidException

key

SOMObject **key** (in SOMObject *element*) ;

Returns the key object of the given element using the Key() method provided by the operations class instance used within the constructor of this collection instance.

lastElement

SOMObject **lastElement** () ;

Returns the last element of the collection.

Precondition: The collection must not be empty.

Exceptions

- IEmptyException

locate

boolean **locate** (in SOMObject *element*,
in ISCursor *cursor*) ;

Locates an element in the collection that is equal to the given element. Sets the cursor to point to the element in the collection, or invalidates the cursor if no such element exists.

If the collection contains several such elements, the first element in iteration order is located.

Precondition: The cursor must belong to the collection.

Return Value: Returns 1 if an element was found.

Exceptions

- `ICursorInvalidException`

locateElementWithKey

boolean **locateElementWithKey** (in `SOMObject key`,
in `ISCursor cursor`) ;

Locates an element in the collection with the same key as the given key. Sets the cursor to point to the element in the collection, or invalidates the cursor if no such element exists.

If the collection contains several such elements, the first element in iteration order is located.

Precondition: The cursor must belong to the collection.

Return Value: Returns 1 if an element was found.

Exceptions

- `ICursorInvalidException`

locateFirst

boolean **locateFirst** (in `SOMObject element`,
in `ISCursor cursor`) ;

Locates the first element in iteration order in the collection that is equal to the given element. Sets the cursor to the located element, or invalidates the cursor if no such element exists.

Precondition: The cursor must belong to the collection.

Return Value: Returns 1 if an element was found.

Exceptions

- `ICursorInvalidException`

locateLast

boolean **locateLast** (in `SOMObject element`,
in `ISCursor cursor`) ;

Locates the last element in iteration order in the collection that is equal to the given element. Sets the cursor to the located element, or invalidates the cursor if no such element exists.

Precondition: The cursor must belong to the collection.

Return Value: Returns 1 if an element was found.

Exceptions

- `ICursorInvalidException`

locateNext

```
boolean locateNext ( in SOMObject element,  
                    in ISCursor cursor ) ;
```

Locates the next element in iteration order in the collection that is equal to the given element, starting at the element next to the one pointed to by the given cursor. Sets the cursor to point to the element in the collection. The cursor is invalidated if the end of the collection is reached and no more occurrences of the given element are left to be visited.

Note: If you code a call to `locateFirst()` and a set of calls to `locateNext()`, each occurrence of an element will be visited exactly once in iteration order.

Precondition: The cursor must belong to the collection and must point to an element of the collection.

Return Value: Returns 1 if an element was found.

Exceptions

- `ICursorInvalidException`

locateNextElementWithKey

```
boolean locateNextElementWithKey (  
    in SOMObject key, in ISCursor cursor ) ;
```

Locates the next element in iteration order in the collection with the given key, starting at the element next to the one pointed to by the given cursor. Sets the cursor to point to the element in the collection. The cursor is invalidated if the end of the collection is reached and no more occurrences of such an element are left to be visited.

Note: If you code a call to `locateFirst()` and a set of calls to `locateNextElementWithKey()`, each occurrence of an element will be visited exactly once in iteration order.

Preconditions: The cursor must belong to the collection and must point to an element of the collection.

Return Value: Returns 1 if an element was found.

Exceptions

- `ICursorInvalidException`

locateOrAdd

boolean **locateOrAdd** (in SOMObject *element*) ;

boolean **locateOrAddWithCursor** (in SOMObject *element*,
in ISCursor *cursor*) ;

Locates an element in the collection that is equal to the given element; See “locate” on page 59 for details on locate(). If no such element is found, locateOrAdd() adds the element as described in “add” on page 48. The cursor is set to the located or added element.

Note: This method may be more efficient than using locate() followed by a conditionally called add().

Preconditions

- The cursor must belong to the collection.
- If the collection is a map or a sorted map and contains an element with the same key as the given element, this element must be equal to the given element.
- The element or key must exist, or
(numberOfElements() < maxNumberOfElements()).

Side Effects: If the element was added, all cursors of this collection, except the given cursor, become undefined.

Return Value: Returns 1 if the element was located. Returns 0 if the element could not be located but had to be added.

Exceptions

- IOutOfCollectionMemoryException
- ICursorInvalidException
- IFullException, if the collection is bounded
- IKeyAlreadyExistsException, if the collection is a map or a sorted map

locateOrAddElementWithKey

boolean **locateOrAddElementWithKey** (
in SOMObject *element*) ;

boolean **locateOrAddElementWithKeyWithCursor** (
in SOMObject *element*; in ISCursor *cursor*) ;

Locates an element in the collection with the given key as described for the locateElementWithKey() function. If no such element exists, locateOrAddElementWithKey() adds the element as described in “add” on page 48. The cursor is set to the located or added element.

Preconditions

- If the collection is bounded and an element with the given key is not already contained, (numberOfElements() < maxNumberOfElements()).
- The cursor must belong to the collection.

Side Effects: If the element was added, all cursors of this collection, except the given cursor, become undefined.

Return Value: Returns 1 if the element was located. Returns 0 if the element could not be located but had to be added.

Exceptions

- `IOutOfCollectionMemoryException`
- `ICursorInvalidException`
- `IFullException`, if the collection is bounded

locatePrevious

boolean **locatePrevious** (in `SOMObject element`,
in `ISCursor cursor`) ;

Locates the previous element in iteration order that is equal to the given element, beginning at the element previous to the one specified by the given cursor and moving in reverse iteration order through the elements. Sets the cursor to the located element, or invalidates the cursor if no such element exists.

Preconditions: The cursor must belong to the collection and must point to an element of the collection.

Return Value: Returns 1 if an element was found.

Exceptions

- `ICursorInvalidException`

maxNumberOfElements

unsigned long **maxNumberOfElements** () ;

Returns the maximum number of elements the collection can contain.

Precondition: The collection is bounded.

Exceptions

- `INotBoundedException`

newCursor

`ISCursor` **newCursor** () ;

Creates an `ISCursor` for the collection and returns a pointer to the cursor. The cursor is initially not valid.

Exceptions

- `IOutOfCollectionMemoryException`

newElementCursor

`ISElementCursor` **newElementCursor** () ;

Creates an `ISElementCursor` for the collection and returns a pointer to the cursor. The cursor is initially not valid.

Exceptions

- `IOutOfCollectionMemoryException`

newOrderedCursor

`ISOrderedCursor newOrderedCursor () ;`

Creates an `ISOrderedCursor` for the collection and returns a pointer to the cursor. The cursor is initially not valid.

Exceptions

- `IOutOfCollectionMemoryException`

notEqual

`boolean notEqual (in INTERF_NAME collection) ;`

Returns 1 if the given collection is not equal to the collection. For a definition of equality for collections, see “equal” on page 58.

numberOfDifferentElements

`unsigned long numberOfDifferentElements () ;`

Returns the number of different elements in the collection.

numberOfDifferentKeys

`unsigned long numberOfDifferentKeys () ;`

Returns the number of different keys in the collection.

numberOfElements

`unsigned long numberOfElements () ;`

Returns the number of elements the collection contains.

numberOfElementsWithKey

`unsigned long numberOfElementsWithKey (
 in SOMObject key) ;`

Returns the number of elements in the collection with the given key.

numberOfOccurrences

`unsigned long numberOfOccurrences (
 in SOMObject element) ;`

Returns the number of occurrences of the given element in the collection.

pop

`void pop () ;`

`void popWithElement (in SOMObject* element) ;`

Copies the last element of the collection to the given element, and removes it from the collection.

Precondition: The collection must not be empty.

Side Effects: All cursors of this collection become undefined.

Exceptions

- `IEmptyException`

position

unsigned long **position** (in `ISCursor cursor`) ;

Determines the position of the current element. Position 1 specifies the first element.

Precondition: The cursor must belong to the collection, and the cursor must point to an element of the collection.

Exceptions

- `ICursorInvalidException`

push

void **push** (in `SOMObject element`) ;

void **pushWithCursor** (in `SOMObject element`,
in `ISCursor cursor`) ;

Adds the element to the collection as the last element (as defined for “add” on page 48), and sets the cursor to the added element.

Preconditions

- The cursor must belong to the collection.
- If the collection is bounded, `(numberOfElements() < maxNumberOfElements())`.

Side Effects: All cursors of this collection, except the given cursor, become undefined.

Exceptions

- `IOutOfCollectionMemoryException`
- `ICursorInvalidException`

remove

boolean **remove** (in `SOMObject element`) ;

Removes an element in the collection that is equal to the given element. If no such element exists, the collection remains unchanged. In collections with nonunique elements, an arbitrary occurrence of the given element will be removed.

Side Effects: If an element was removed, all cursors of this collection become undefined.

Return Value: Returns 1 if an element was removed.

removeAll

void **removeAll** () ;

Removes all elements from the collection.

Exceptions

- IPredicateOverrideException
- IUserPredicateException

Side Effects: All cursors of this collection become undefined.

removeAllWithPredicate

unsigned long **removeAllWithPredicate** (
in ISPredicate predicate) ;

Removes all elements from this collection for which the given property function returns 1.

Side Effects: If any elements were removed, all cursors of this collection become undefined.

Return Value: The number of elements removed.

removeAllElementsWithKey

unsigned long **removeAllElementsWithKey** (in SOMObject key) ;

Removes all elements from the collection with the same key as the given key.

Side Effects: If any elements were removed, all cursors of this collection become undefined.

Return Value: The number of elements removed.

removeAllOccurrences

unsigned long **removeAllOccurrences** (in SOMObject element) ;

Removes all elements from the collection that are equal to the given element, and returns the number of elements removed.

Side Effects: If any elements were removed, all cursors of this collection become undefined.

removeAt

void **removeAt** (in ISCursor cursor) ;

Removes the element pointed to by the given cursor. The given cursor is invalidated.

Preconditions: The cursor must belong to the collection and must point to an element of the collection.

Side Effects: All cursors of this collection, except the given cursor, become undefined.

Exceptions

- `ICursorInvalidException`

removeAtPosition

`void removeAtPosition (in unsigned long position) ;`

Removes the element from the collection that is at the given position.

The first element of the collection has position 1.

Precondition: Position must be a valid position in the collection; that is, $(1 \leq \textit{position} \leq \text{numberOfElements}())$.

Side Effects: All cursors of this collection become undefined.

Exceptions

- `IPositionInvalidException`

removeElementWithKey

`boolean removeElementWithKey (in SOMObject key) ;`

Removes an element from the collection with the same key as the given key. If no such element exists, the collection remains unchanged. In collections with nonunique elements, an arbitrary occurrence of such an element will be removed.

Side Effects: If an element was removed, all cursors of this collection become undefined.

Return Value: Returns 1 if an element was removed.

removeFirst

`void removeFirst () ;`

Removes the first element from the collection.

Precondition: The collection must not be empty.

Side Effects: All cursors of this collection become undefined.

Exceptions

- `IEmptyException`

removeLast

`void removeLast () ;`

Removes the last element from the collection.

Precondition: The collection must not be empty.

Side Effects: All cursors of this collection become undefined.

Exceptions

- IEmptyException

replaceAt

```
void replaceAt ( in ISCursor cursor,  
                in SOMObject element ) ;
```

Replaces the element pointed to by the cursor with the given element.

Preconditions

- The cursor must belong to the collection and must point to an element of the collection.
- The given element must have the same positioning property as the replaced element.

Exceptions

- ICursorInvalidException
- IInvalidReplacementException

replaceElementWithKey

```
boolean replaceElementWithKey ( in SOMObject element ) ;
```

```
boolean replaceElementWithKeyWithCursor ( in SOMObject element,  
                                           in ISCursor cursor ) ;
```

Replaces an element with the same key as the given element by the given element, and sets the cursor to this element. If no such element exists, it invalidates the cursor. In collections with nonunique elements, an arbitrary occurrence of such an element will be replaced.

Precondition: The cursor must belong to the collection.

Return Value: Returns 1 if an element was replaced.

Exceptions

- ICursorInvalidException

setToFirst

```
boolean setToFirst ( in ISCursor cursor ) ;
```

Sets the cursor to the first element of the collection in iteration order. If the collection is empty (if no first element exists), it invalidates the given cursor.

Depending on the final language used this method must be used in its fully qualified version because the Cursor interface supports the same method.

Precondition: The cursor must belong to the collection.

Return Value: Returns 1 if the collection is not empty.

Exceptions

- `ICursorInvalidException`

setToLast

boolean **setToLast** (in `ISCursor cursor`) ;

Sets the cursor to the last element of the collection in iteration order. If the collection is empty (if no last element exists), the given cursor is no longer valid.

Depending on the final language used this method must be used in its fully qualified version because the `Cursor` interface supports the same method.

Precondition: The cursor must belong to the collection.

Return Value: Returns 1 if the collection is not empty.

Exceptions

- `ICursorInvalidException`

setToNext

boolean **setToNext** (in `ISCursor cursor`) ;

Sets the cursor to the next element in the collection in iteration order. If no more elements are left to be visited, the given cursor will no longer be valid.

Depending on the final language used this method must be used in its fully qualified version because the `Cursor` interface supports the same method.

Precondition: The cursor must belong to the collection and must point to an element.

Return Value: Returns 1 if there is a next element.

Exceptions

- `ICursorInvalidException`

setToNextDifferentElement

boolean **setToNextDifferentElement** (
 in `ISCursor cursor`) ;

Sets the cursor to the next element in iteration order in the collection that is different from the element pointed to by the given cursor. If no more elements are left to be visited, the given cursor will no longer be valid.

Precondition: The cursor must belong to the collection and must point to an element of the collection.

Return Value: Returns 1 if a subsequent element was found that is different.

Exceptions

- `ICursorInvalidException`

setToNextWithDifferentKey

boolean **setToNextWithDifferentKey** (in ISCursor *cursor*) ;

Sets the cursor to the next element in the collection in iteration order with a key different from the key of the element pointed to by the given cursor. If no such element exists, the given cursor is no longer valid.

Preconditions: The cursor must belong to the collection and must point to an element of the collection.

Return Value: Returns 1 if a subsequent element was found whose key is different from the current key.

Exceptions

- ICursorInvalidException

setPosition

void **setPosition** (in unsigned long *position*,
in ISCursor *cursor*) ;

Sets the cursor to the element at the given position. Position 1 specifies the first element.

Precondition

- The cursor must belong to the collection.
- Position must be a valid position in the collection; that is, $(1 \leq position \leq numberOfElements())$.

Exceptions

- ICursorInvalidException
- IPositionInvalidException

setToPrevious

boolean **setToPrevious** (in ISCursor *cursor*) ;

Sets the cursor to the previous element in iteration order, or invalidates the cursor if no such element exists.

Depending on the final language used this method must be used in its fully qualified version because the Cursor interface supports the same method.

Preconditions: The cursor must belong to the collection and must point to an element of the collection.

Return Value: Returns 1 if a previous element exists.

Exceptions

- ICursorInvalidException

sort

```
void sort ( in ISComparator comparator ) ;
```

Sorts the collection so that the elements occur in ascending order. The relation of two elements is defined by the *compare* method, which you provide when subclassing from *ISComparator*.

Note: The *Compare* method must deliver a result according to the following rules:

```
>0      if (element1 > element2)
0       if (element1 == element2)
<0      if (element1 < element2)
```

Side Effects: All cursors of this collection become undefined.

top

```
SOMObject top ( ) ;
```

Returns the last SOMObject element of the collection.

Precondition: The collection must not be empty.

Exceptions

- IEmptyException

unionWith

```
void unionWith ( in INTERF_NAME collection ) ;
```

Makes the collection the union of the collection and the given collection. The *union* of A and B is the set of elements that are members of A or B or both.

The following rule applies for bags with duplicate elements: If bag P contains the element X *m* times and bag Q contains the element X *n* times, the *union* of P and Q contains the element X *m+n* times.

Preconditions: Because the elements from the given collection are added to the collection one by one, the following preconditions are tested for each individual add operation :

- If the collection is bounded and unique, the element or key must exist or (numberOfElements() < maxNumberOfElements()).
- If the collection is bounded and nonunique, (numberOfElements() < maxNumberOfElements()).
- If the collection is a map or a sorted map and contains an element with the same key as the given element, this element must be equal to the given element.

Side Effects: If any elements were added to the collection, all cursors of this collection become undefined.

Exceptions

- IOutOfCollectionMemoryException
- IFullException, if the collection is bounded
- IKeyAlreadyExistsException, if the collection is a map or a sorted map

Chapter 9. Bag

A *Bag* is an unordered collection of zero or more elements with no key. Multiple elements are supported. A request to add an element that already exists is not ignored.

Figure 4 on page 26 gives an overview of the properties of a Bag and its relationship to other flat collections.

An example of using a Bag is a program for entering observations on species of plants and animals found in a river. Each time you spot a plant or an animal in the river, you enter the name of the species into the collection. If you spot a species twice during an observation period, the species is added twice, because a Bag supports multiple elements. You can locate the name of a species that you have observed, and you can determine the number of observations of that species; however, you cannot sort the collection by species (because a Bag is an unordered collection). To sort the elements of a Bag you should use a Sorted Bag instead.

The following rule applies for duplicates: If Bag P contains the element X m times and Bag Q contains the element X n times, then the *union* of P and Q contains the element X $m+n$ times, the *intersection* of P and Q contains the element X $\text{MIN}(m,n)$ times, and the *difference* of P and Q contains the element X $m-n$ times if m is $> n$, and *zero* times if m is $\leq n$.

Derivation

Collection
Equality Collection
Bag

Interface Name	Filestem
ISBag	sbag

Members

All member functions of flat collections are described in Chapter 7, "Introduction to Flat Collections" on page 45. The following members are provided for Bag:

Method	Page	Method	Page
add	48	isBounded	58
addAllFrom	48	isEmpty	58
addDifference	51	isFull	59
addIntersection	51	locate	59
addUnion	53	locateNext	61
allElementsDo	53	locateOrAdd	62
anyElement	54	maxNumberOfElements	63
assign	54	newCursor	63
contains	55	newElementCursor	63
containsAllFrom	55	notEqual	64
copy	55	numberOfDifferentElements	64
Destructor	47	numberOfElements	64
differenceWith	56	numberOfOccurrences	64
elementAt	56	remove	65
equal	58	removeAllOccurrences	66
Initializer Method	47	removeAll	66
intersectionWith	58	removeAt	66

Bag

Method	Page	Method	Page
replaceAt	68	setToNextDifferentElement	69
setToFirst	68	unionWith	71
setToNext	69		

You can use an `ISElementCursor` with a `Bag`. The members for `ISElementCursor` are described in Chapter 29, “Cursor” on page 115.

Required Operations

For `ISBag`, the operations listed below are required for the element type. You can either use the default operations from `ISOps` or override them with your own implementation.

Element Type

- `Assign()`
- `Equal()`
- `Compare()`

Chapter 10. Deque

A *Deque* or double-ended queue is a sequence with restricted access. It is an ordered collection of elements with no key and no element equality. The elements are arranged so that each collection has a first and a last element, each element except the last has a next element, and each element but the first has a previous element. You can only add or remove the first or last element.

The type and value of the elements are irrelevant, and have no effect on the behavior of the collection.

An example of using a Deque is a program for managing a lettuce warehouse. Cases of lettuce arriving into the warehouse are registered at one end of the queue (the “fresh” end) by the receiving department. The shipping department reads the other end of the queue (the “old” end) to determine which case of lettuce to ship next. If an order was to come in for very fresh lettuce, which is sold at a premium, the shipping department reads the “fresh” end of the queue to select the freshest case of lettuce available.

Derivation

```
Collection
  Ordered Collection
    Sequential Collection
      Sequence
        Deque
```

Note that Deque is based on sequence but is not actually. See “Restricted Access” for further details.

Interface Name

Filestem

ISDeque

sdqu

Members

All members of flat collections are described in Chapter 7, “Introduction to Flat Collections” on page 45. The following members are provided for Deque:

Method	Page	Method	Page
add	48	isFull	59
addAllFrom	48	isLast	59
addAsFirst	49	lastElement	59
addAsLast	49	maxNumberOfElements	63
allElementsDo	53	newCursor	63
anyElement	54	newElementCursor	63
assign	54	newOrderedCursor	64
compare	54	numberOfElements	64
copy	55	position	65
Destructor	47	removeAll	66
elementAt	56	removeFirst	67
elementAtPosition	56	removeLast	67
firstElement	58	setToFirst	68
Initializer Method	47	setToLast	69
isBounded	58	setToNext	69
isEmpty	58	setToPosition	70
isFirst	59	setToPrevious	70

You can use an `ISOrderedCursor` with a `Deque`. The members for `ISOrderedCursor` are described in Chapter 29, “Cursor” on page 115.

Required Operations

For `ISDeque`, the operations listed below are required for the element type. You can either use the default operations from `ISOps` or override them with your own implementation.

Element Type

- `Assign()`

A coding example for a *Deque* is provided in the appendix in “Coding Example for Deque” on page 149.

Chapter 11. Equality Sequence

An *Equality Sequence* is an ordered collection of elements. The elements are arranged so that each collection has a first and a last element, each element except the last has a next element, and each element but the first has a previous element. An Equality Sequence supports element equality, which gives you the ability, for example, to search for particular elements.

An example of using an Equality Sequence is a program that calculates members of the Fibonacci sequence and places them in a collection with multiple elements of the same value being allowed. For example, the sequence begins with two instances of the value 1. You can search for a given element, for example 8, and find out what element follows it in the sequence. Element equality allows you to accomplish this using the `locate()` and `setToNext()` functions.

Derivation

Collection

Equality Collection

Sequential Collection

Equality Sequence

Figure 2 on page 19 illustrates the properties of an Equality Sequence and its relationship to other flat collections.

Interface Name	Filestem
ISEqualitySequence	ses

Members

All members of flat collections are described in Chapter 7, "Introduction to Flat Collections" on page 45. The following members are provided for Equality Sequence:

Method	Page	Method	Page
add	48	isEmpty	58
addAllFrom	48	isFirst	59
addAsFirst	49	isFull	59
addAsLast	49	isLast	59
addAsNext	50	lastElement	59
addAsPrevious	50	locate	59
addAtPosition	50	locateFirst	60
allElementsDo	53	locateLast	60
anyElement	54	locateNext	61
assign	54	locateOrAdd	62
compare	54	locatePrevious	63
contains	55	maxNumberOfElements	63
containsAllFrom	55	newCursor	63
copy	55	newElementCursor	63
Destructor	47	newOrderedCursor	64
elementAt	56	notEqual	64
elementAtPosition	56	numberOfElements	64
equal	58	numberOfOccurrences	64
firstElement	58	position	65
Initializer Method	47	remove	65
isBounded	58	removeAll	66

Equality Sequence

Method	Page	Method	Page
removeAllOccurrences	66	setToNext	69
removeAt	66	setToPosition	70
removeAtPosition	67	setToPrevious	70
removeFirst	67	sort	71
removeLast	67		
replaceAt	68		
setToFirst	68		
setToLast	69		

You can use an `ISOrderedCursor` with an Equality Sequence. The members for `ISOrderedCursor` are described in Chapter 29, “Cursor” on page 115.

Required Operations

For `ISEqualitySequence`, the operations listed below are required for the element type. You can either use the default operations from `ISOps` or override them with your own implementation.

Element Type

- `Assign()`
- `Equal()`

Chapter 12. Heap

A *Heap* is an unordered collection of zero or more elements with no key. Element equality is not supported, while multiple elements are supported. The type and value of the elements are irrelevant, and have no effect on the behavior of the Heap.

You can compare using a Heap collection to managing the scrap metal entering a scrapyard. Pieces of scrap are placed in the Heap in an arbitrary location, and an element can be added multiple times (for example, you could have a rear left fender from a particular kind of car). When a customer requests a certain amount of scrap, elements are removed from the Heap in an arbitrary order until the required amount is reached. You cannot search for a specific piece of scrap except by examining each piece of scrap in the Heap and manually comparing it to the piece you are looking for.

Figure 2 on page 19 illustrates the properties of a Heap and its relationship to other flat collections.

Derivation

Collection
Heap

Interface Name	Filestem
ISHeap	shp

Members

All members of flat collections are described in Chapter 7, "Introduction to Flat Collections" on page 45. The following members are provided for Heap:

Method	Page	Method	Page
add	48	maxNumberOfElements	63
addAllFrom	48	newCursor	63
allElementsDo	53	newElementCursor	63
anyElement	54	numberOfElements	64
assign	54	removeAll	66
copy	55	removeAt	66
Destructor	47	replaceAt	68
elementAt	56	setToFirst	68
Initializer Method	47	setToNext	69
isBounded	58		
isEmpty	58		
isFull	59		

You can use an *ISElementCursor* with a Heap. The members for *ISElementCursor* are described in Chapter 29, "Cursor" on page 115.

Required Operations

For ISHeap, the operations listed below are required for the element type. You can either use the default operations from ISOps or override them with your own implementation.

Element Type

- Assign()

Chapter 13. Key Bag

A *key bag* is an unordered collection of zero or more elements that have a key. Multiple elements are supported.

An example of using a Key Bag is a program that manages the distribution of combination locks to members of a fitness club. The element key is the number that is printed on the back of each combination lock. Each element also has data members for the club member's name, membership number, and so on. When you join the club, you are given one of the available combination locks, and your name, membership number, and the number on the combination lock are entered into the collection. Because a given number on a combination lock may appear on several locks, the program allows the same lock number to be added to the collection multiple times. When you return a lock because you are leaving the club, the program finds each element whose key matches your lock's serial number, and deletes one such element that has your name associated with it.

Figure 3 on page 23 illustrates the differences in behavior between map, relation, key set, and key bag when identical elements and elements with the same key are added.

Derivation

Collection

Key Collection

Key Bag

Figure 2 on page 19 gives an overview of the properties of a key bag and its relationship to other flat collections.

Interface Name

Filestem

ISKeyBag

skb

Members

All members of flat collections are described in Chapter 7, "Introduction to Flat Collections" on page 45. The following members are provided for key bag:

Method	Page	Method	Page
add	48	locateElementWithKey	60
addAllFrom	48	locateNextElementWithKey	61
addOrReplaceElementWithKey	52	locateOrAddElementWithKey	62
allElementsDo	53	maxNumberOfElements	63
anyElement	54	newCursor	63
assign	54	newElementCursor	63
containsAllKeysFrom	55	numberOfDifferentKeys	64
containsElementWithKey	55	numberOfElements	64
copy	55	numberOfElementsWithKey	64
Destructor	47	removeAll	66
elementAt	56	removeAllElementsWithKey	66
elementWithKey	57	removeAt	66
Initializer Method	47	removeElementWithKey	67
isBounded	58	replaceAt	68
isEmpty	58	replaceElementWithKey	68
isFull	59	setToFirst	68
key	59	setToNext	69

Key Bag

Method	Page
setToNextWithDifferentKey	70

You can use an `ISElementCursor` with a `KeyBag`. The members for `ISElementCursor` are described in Chapter 29, “Cursor” on page 115.

Required Operations

For `ISKeyBag`, the operations listed below are required for the element type and key type. You can either use the default operations from `ISOps` or override them with your own implementation.

Element Type

- `Assign()`
- `Key()`

Key Type

- `KeyEqual`
- `KeyHash`

A coding example for a *key bag* is provided in the appendix in “Coding Example for Key Bag” on page 156.

Chapter 14. Key Set

A *Key Set* is an unordered collection of zero or more elements that have a key; element equality is not supported and only unique elements are supported, in terms of their key.

An example of using a Key Set is a program that allocates rooms to patrons checking into a hotel. The room number serves as the element's key, and the patron's name is a data member of the element. When you check in at the front desk, the clerk pulls a room key from the board, and enters that key's number and your name into the collection. When you return the key at check-out time, the record for that key is removed from the collection. You cannot add an element to the collection that is already present, because there is only one key for each room. If you attempt to add an element that is already present, the `add()` function returns `false` to indicate that the element was not added.

Figure 3 on page 23 illustrates the differences in behavior between map, relation, key set, and key bag when identical elements and elements with the same key are added.

Figure 2 on page 19 gives an overview of the properties of a Key Set and its relationship to other flat collections.

Derivation

```
Collection
  Key Collection
    Key Set
```

Interface Name	Filestem
ISKeySet	sks

Members

All members of flat collections are described in Chapter 7, "Introduction to Flat Collections" on page 45. The following members are provided for Key Set:

Method	Page	Method	Page
<code>add</code>	48	<code>locateOrAddElementWithKey</code>	62
<code>addAllFrom</code>	48	<code>maxNumberOfElements</code>	63
<code>addOrReplaceElementWithKey</code>	52	<code>newCursor</code>	63
<code>allElementsDo</code>	53	<code>newElementCursor</code>	63
<code>anyElement</code>	54	<code>numberOfElements</code>	64
<code>assign</code>	54	<code>removeAll</code>	66
<code>containsAllKeysFrom</code>	55	<code>removeAt</code>	66
<code>containsElementWithKey</code>	55	<code>removeElementWithKey</code>	67
<code>copy</code>	55	<code>replaceAt</code>	68
<code>Destructor</code>	47	<code>replaceElementWithKey</code>	68
<code>elementAt</code>	56	<code>setToFirst</code>	68
<code>elementWithKey</code>	57	<code>setToNext</code>	69
<code>Initializer Method</code>	47		
<code>isBounded</code>	58		
<code>isEmpty</code>	58		
<code>isFull</code>	59		
<code>key</code>	59		
<code>locateElementWithKey</code>	60		

You can also an ISElementCursor with a KeySet. The members for ISElementCursor are described in Chapter 29, “Cursor” on page 115.

Required Operations

For ISKeySet, the operations listed below are required for the element type and key type. You can either use the default operations from ISOps or override them with your own implementation.

Element Type

- Assign()
- Key()

Key Type

- KeyCompare

Chapter 15. Key Sorted Bag

A *Key Sorted Bag* is an ordered collection of zero or more elements that have a key. Elements are sorted according to the value of their key field; element equality is not supported while multiple elements are.

An example of using a Key Sorted Bag is a program that maintains a list of families, sorted by the number of family members in each family. The key is the number of family members. You can add an element whose key is already in the collection (because two families can have the same number of members), and you can generate a list of families sorted by size; however, you cannot locate a family except by its key, because a Key Sorted Bag does not support element equality.

Figure 2 on page 19 gives an overview of the properties of a Key Sorted Bag and its relationship to other flat collections.

Derivation



Interface Name	Filestem
ISKeySortedBag	sksb

Members

All members of flat collections are described in Chapter 7, "Introduction to Flat Collections" on page 45. The following members are provided for Key Sorted Bag:

Method	Page	Method	Page
add	48	locateElementWithKey	60
addAllFrom	48	locateNextElementWithKey	61
addOrReplaceElementWithKey	52	locateOrAddElementWithKey	62
allElementsDo	53	maxNumberOfElements	63
anyElement	54	newCursor	63
assign	54	newElementCursor	63
compare	54	newOrderedCursor	64
containsAllKeysFrom	55	numberOfDifferentKeys	64
containsElementWithKey	55	numberOfElements	64
copy	55	numberOfElementsWithKey	64
Destructor	47	position	65
elementAt	56	removeAll	66
elementAtPosition	56	removeAllElementsWithKey	66
elementWithKey	57	removeAt	66
firstElement	58	removeAtPosition	67
Initializer Method	47	removeElementWithKey	67
isBounded	58	removeFirst	67
isEmpty	58	removeLast	67
isFirst	59	replaceAt	68
isFull	59	replaceElementWithKey	68
isLast	59	setToFirst	68
key	59	setToLast	69
lastElement	59	setToNext	69

Key Sorted Bag

Method	Page	Method	Page
setToNextWithDifferentKey	70	setToPrevious	70
setPosition	70		

You can use an `ISOrderedCursor` with a `KeySorted Bag`. The members for `ISOrderedCursor` are described in Chapter 29, “Cursor” on page 115.

Required Operations

For `ISKeySortedBag`, the operations listed below are required for the element type and key type. You can either use the default operations from `ISOps` or override them with your own implementation.

Element Type

- `Assign()`
- `Key()`

Key Type

- `KeyCompare`

Chapter 16. Key Sorted Set

A *Key Sorted Set* is an ordered collection of zero or more elements that have a key. Elements are sorted according to the value of their key field. Element equality is not supported and only elements with unique keys are supported; requests to add an element whose key already exists are ignored.

An example of using a Key Sorted Set is a program that keeps track of canceled credit card numbers and the individuals to whom they are issued. Each card number occurs only once, and the collection is sorted by card number. When a merchant enters a customer's card number into a point-of-sale terminal, the collection is checked to see if that card number is listed in the collection of canceled cards. If it is found, the name of the individual is shown, and the merchant is given directions for contacting the credit card company. If the card number is not found, the transaction can proceed because the card is considered to be valid. A list of canceled cards is printed out each month, sorted by card number, and distributed to all merchants who do not have an automatic point-of-sale terminal installed.

Figure 2 on page 19 gives an overview of the properties of a Key Sorted Set and its relationship to other flat collections.

Derivation



Interface Name

Filestem

ISKeySortedSet

skss

Members

All members of flat collections are described in Chapter 7, "Introduction to Flat Collections" on page 45. The following members are provided for Key Sorted Set:

Method	Page	Method	Page
add	48	isFirst	59
addAllFrom	48	isFull	59
addOrReplaceElementWithKey	52	isLast	59
allElementsDo	53	key	59
anyElement	54	lastElement	59
assign	54	locateElementWithKey	60
compare	54	locateNextElementWithKey	61
containsAllKeysFrom	55	locateOrAddElementWithKey	62
containsElementWithKey	55	numberOfElements	63
copy	55	newCursor	63
Destructor	47	newElementCursor	63
elementAt	56	newOrderedCursor	64
elementAtPosition	56	numberOfElements	64
elementWithKey	57	position	65
firstElement	58	removeAll	66
Initializer Method	47	removeAt	66
isBounded	58	removeAtPosition	67
isEmpty	58	removeElementWithKey	67

Key Sorted Set

Method	Page	Method	Page
removeFirst	67	setToNext	69
removeLast	67	setToPosition	70
replaceAt	68	setToPrevious	70
replaceElementWithKey	68		
setToFirst	68		
setToLast	69		

You can use an `ISOrderedCursor` with a `KeySortedSet`. The members for `ISOrderedCursor` are described in Chapter 29, “Cursor” on page 115.

Required Operations

For `ISKeySortedSet`, the operations listed below are required for the element type and key type. You can either use the default operations from `ISOps` or override them with your own implementation.

Element Type

- `Assign()`
- `Key()`

Key Type

- `KeyCompare`

Chapter 17. Map

A *Map* is an unordered collection of zero or more elements that have a key. Element equality is supported and the values of the elements are relevant.

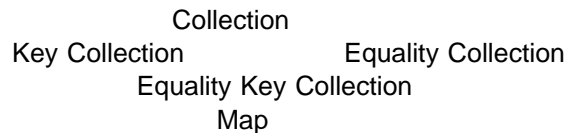
Only elements with unique keys are supported. A request to add an element whose key already exists in another element of the collection causes an exception to be thrown and the request to add a duplicate element is ignored.

An example of using a Map is a program that translates integer values between the ranges of 0 and 20 to their written equivalents, or between written numbers and their numeric values. Two Maps are created, one with the integer values as keys, one with the written equivalents as keys. You can enter a number, and that number is used as a key to locate the written equivalent. You can enter a written equivalent of a number, and that text is used as a key to locate the value. A given key always matches only one element. You cannot add an element with a key of 1 or "one" if that element is already present in the collection.

Figure 3 on page 23 illustrates the differences in behavior between Map, relation, key set, and key bag when identical elements and elements with the same key are added.

Figure 2 on page 19 gives an overview of the properties of a Map and its relationship to other flat collections.

Derivation



Interface Name	Filestem
ISMap	smap

Members

All members of flat collections are described in Chapter 7, "Introduction to Flat Collections" on page 45. The following members are provided for Map:

Method	Page	Method	Page
add	48	differenceWith	56
addAllFrom	48	elementAt	56
addDifference	51	elementWithKey	57
addIntersection	51	equal	58
addOrReplaceElementWithKey	52	Initializer Method	47
addUnion	53	intersectionWith	58
allElementsDo	53	isBounded	58
anyElement	54	isEmpty	58
assign	54	isFull	59
contains	55	key	59
containsAllFrom	55	locate	59
containsAllKeysFrom	55	locateElementWithKey	60
containsElementWithKey	55	locateOrAdd	62
copy	55	locateOrAddElementWithKey	62
Destructor	47	maxNumberOfElements	63

Map

Method	Page	Method	Page
newCursor	63	replaceAt	68
newElementCursor	63	replaceElementWithKey	68
notEqual	64	setToFirst	68
numberOfElements	64	setToNext	69
remove	65	unionWith	71
removeAll	66		
removeAt	66		
removeElementWithKey	67		

You can use an `ISElementCursor` with a `SortedMap`. The members for `ISElementCursor` are described in Chapter 29, “Cursor” on page 115.

Required Operations

For `ISMap`, the operations listed below are required for the element type and key type. You can either use the default operations from `ISOps` or override them with your own implementation.

Element Type

- `Assign()`
- `Equal()`
- `Key()`

Key Type

- `KeyCompare`

Chapter 18. Priority Queue

A *Priority Queue* is a key sorted bag with restricted access. It is an ordered collection of zero or more elements. Keys and multiple elements are supported and element equality is not supported.

When an element is added, it is placed in the queue according to its key value or *priority*. The highest priority is indicated by the lowest key value and you can only remove the element with the highest priority. Within the Priority Queue, elements are sorted according to ascending key values, as in other key collections. You can only remove the element with the lowest key value.

For elements with equal priority, the Priority Queue has a first-in, first-out behavior.

An example of a Priority Queue is a program used to assign priorities to service calls in a heating repair firm. When a customer calls with a problem, a record with the customer's name and the seriousness of the situation is placed in a Priority Queue. When a service person becomes available, customers are chosen by the program beginning with those whose situation is most severe. In this example, a serious problem such as a nonfunctioning furnace would be indicated by a low value for the priority, and a minor problem such as a noisy radiator would be indicated by a high value for the priority.

Derivation

Key Sorted Collection
 Key Sorted Bag
 Priority Queue

Note that Priority Queue is based on key sorted bag but is not actually derived from it or from the other classes shown above. The diagram does not show all bases of Priority Queue. See Figure 4 on page 26 for a complete illustration. See "Restricted Access" for further details.

Interface Name	Filestem
ISPriorityQueue	spqu

Members

All members of flat collections are described in Chapter 7, "Introduction to Flat Collections" on page 45. The following members are provided for Priority Queue:

Method	Page	Method	Page
add	48	elementAtPosition	56
addAllFrom	48	elementWithKey	57
allElementsDo	53	enqueue	57
anyElement	54	firstElement	58
assign	54	Initializer Method	47
compare	54	isBounded	58
containsAllKeysFrom	55	isEmpty	58
containsElementWithKey	55	isFirst	59
copy	55	isFull	59
Destructor	47	isLast	59
dequeue	56	key	59
elementAt	56	lastElement	59

Priority Queue

Method	Page	Method	Page
locateElementWithKey	60	setToLast	69
locateNextElementWithKey	61	setToNext	69
locateOrAddElementWithKey	62	setToNextWithDifferentKey	70
maxNumberOfElements	63	setToPosition	70
newCursor	63	setToPrevious	70
newElementCursor	63		
newOrderedCursor	64		
numberOfDifferentKeys	64		
numberOfElements	64		
numberOfElementsWithKey	64		
position	65		
removeAll	66		
removeFirst	67		
setToFirst	68		

You can use an `ISOrderedCursor` with a `PriorityQueue`. The members for `ISOrderedCursor` are described in Chapter 29, “Cursor” on page 115.

Required Operations

For `ISPriorityQueue`, the operations listed below are required for the element type and key type. You can either use the default operations from `ISOps` or override them with your own implementation.

Element Type

- `Assign()`
- `Key()`

Key Type

- `KeyCompare`

Chapter 19. Queue

A *Queue* is a sequence with restricted access. It is an ordered collection of elements with no key and no element equality. The elements are arranged so that each collection has a first and a last element, each element except the last has a next element, and each element but the first has a previous element. The type and value of the elements are irrelevant, and have no effect on the behavior of the collection.

You can only add an element as the last element, and you can only remove the first element. Consequently, the elements of a Queue are in chronological order.

A Queue is characterized by a first-in, first-out (FIFO) behavior.

An example of using a Queue is a program that processes requests for parts at the cash sales desk of a warehouse. A request for a part is added to the Queue when the customer's order is taken, and is removed from the Queue when an order picker receives the order form for the part. Using a Queue collection in such an application ensures that all orders for parts are processed on a first-come first-served basis.

Derivation

```
Collection
  Ordered Collection
    Sequential Collection
      Sequence
        Queue
```

Note that Queue is based on sequence but is not actually derived from it or from the other classes shown above. See "Restricted Access" for further details.

Interface Name	Filestem
ISQueue	squ

Members

All members of flat collections are described in Chapter 7, "Introduction to Flat Collections" on page 45. The following members are provided for Queue:

Method	Page	Method	Page
add	48	isBounded	58
addAllFrom	48	isEmpty	58
addAsLast	49	isFirst	59
allElementsDo	53	isFull	59
anyElement	54	isLast	59
assign	54	lastElement	59
compare	54	maxNumberOfElements	63
copy	55	newCursor	63
dequeue	56	newElementCursor	63
Destructor	47	newOrderedCursor	64
elementAt	56	numberOfElements	64
elementAtPosition	56	position	65
enqueue	57	removeAll	66
firstElement	58	removeFirst	67
Initializer Method	47	setToFirst	68

Queue

Method	Page	Method	Page
setToLast	69	setToNext	69
		setToPosition	70
		setToPrevious	70

You can use an `ISOrderedCursor` with a `Queue`. The members for `ISOrderedCursor` are described in Chapter 29, “Cursor” on page 115.

Required Operations

For `ISQueue`, the operations listed below are required for the element type. You can either use the default operations from `ISOps` or override them with your own implementation.

Element Type

- `Assign()`

Chapter 20. Relation

A *Relation* is an unordered collection of zero or more elements that have a key. Element equality is supported, and the values of the elements are relevant.

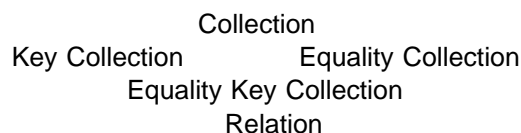
The keys of the elements are not unique. You can add an element whether or not there is already an element in the collection with the same key.

Figure 3 on page 23 illustrates the differences in behavior between map, Relation, key set, and key bag when identical elements and elements with the same key are added.

An example of using a Relation is a program that maintains a list of all your relatives, with an individual's relationship, to you, as the key. You can add an aunt, uncle, grandmother, daughter, father-in-law, and so on. You can add an aunt even if an aunt is already in the collection, because you can have several relatives who have the same relationship to you. (For unique relationships such as mother or father, your program would have to check the collection to make sure it did not already contain a family member with that key, before adding the family member.) You can locate a member of the family, but the family members are not in any particular order.

Figure 2 on page 19 gives an overview of the properties of a relation and its relationship to other flat collections.

Derivation



Interface Name	Filestem
ISRelation	srel

Members

All members of flat collections are described in Chapter 7, "Introduction to Flat Collections" on page 45. The following members are provided for relation:

Method	Page	Method	Page
add	48	elementAt	56
addAllFrom	48	elementWithKey	57
addDifference	51	equal	58
addIntersection	51	Initializer Method	47
addOrReplaceElementWithKey	52	intersectionWith	58
addUnion	53	isBounded	58
allElementsDo	53	isEmpty	58
anyElement	54	isFull	59
assign	54	key	59
contains	55	locate	59
containsAllFrom	55	locateElementWithKey	60
containsAllKeysFrom	55	locateNextElementWithKey	61
containsElementWithKey	55	locateOrAdd	62
copy	55	locateOrAddElementWithKey	62
Destructor	47	maxNumberOfElements	63
differenceWith	56	newCursor	63

Relation

Method	Page	Method	Page
newElementCursor	63	removeAt	66
notEqual	64	removeElementWithKey	67
numberOfDifferentKeys	64	replaceAt	68
numberOfElements	64	replaceElementWithKey	68
numberOfElementsWithKey	64	setToFirst	68
remove	65	setToNext	69
removeAll	66	setToNextWithDifferentKey	70
removeAllElementsWithKey	66	unionWith	71

You can use an `ISElementCursor` with a `Relation`. The members for `ISElementCursor` are described in Chapter 29, “Cursor” on page 115.

Required Operations

For `ISRelation`, the operations listed below are required for the element type and key type. You can either use the default operations from `ISOps` or override them with your own implementation.

Element Type

- `Assign()`
- `Equal()`
- `Key()`

Key Type

- `KeyEqual`
- `KeyHash`

Chapter 21. Sequence

A *Sequence* is an ordered collection of elements. The elements are arranged so that each collection has a first and a last element, each element except the last has a next element, and each element but the first has a previous element.

The type and value of the elements are irrelevant, and have no effect on the behavior of the collection. Elements can be added and deleted from any position in the collection. Elements can be retrieved or replaced. A Sequence does not support element equality or a key. If you require element equality for a Sequence, you can use an equality Sequence. See Chapter 11, "Equality Sequence" on page 77 for further details.

An example of a Sequence is a program that maintains a list of the words in a paragraph. The order of the words is obviously important, and you can add or remove words at a given position, but you cannot search for individual words except by iterating through the collection and comparing each word to the word you are searching for. You can add a word that is already present in the Sequence, because a given word may be used more than once in a paragraph.

Figure 2 on page 19 illustrates the properties of a LIST and its relationship to other flat collections.

Derivation

```
Collection
  Ordered Collection
    Sequential Collection
      Sequence
```

Interface Name

Filestem

ISSequence

sseq

Members

All members of flat collections are described in Chapter 7, "Introduction to Flat Collections" on page 45. The following members are provided for LIST:

Method	Page	Method	Page
add	48	isBounded	58
addAllFrom	48	isEmpty	58
addAsFirst	49	isFirst	59
addAsLast	49	isFull	59
addAsNext	50	isLast	59
addAsPrevious	50	lastElement	59
addAtPosition	50	maxNumberOfElements	63
allElementsDo	53	newCursor	63
anyElement	54	newElementCursor	63
assign	54	newOrderedCursor	64
compare	54	numberOfElements	64
copy	55	position	65
Destructor	47	removeAll	66
elementAt	56	removeAt	66
elementAtPosition	56	removeAtPosition	67
firstElement	58	removeFirst	67
Initializer Method	47	removeLast	67

Sequence

Method	Page	Method	Page
replaceAt	68	setPosition	70
setToFirst	68	setToPrevious	70
setToLast	69	sort	71
setToNext	69		

You can use an `ISOrderedCursor` with a `Sequence`. The members for `ISOrderedCursor` are described in Chapter 29, “Cursor” on page 115.

Required Operations

For `ISSequence`, the operations listed below are required for the element type and key type. You can either use the default operations from `ISOps` or override them with your own implementation.

Element Type

- `Assign()`

Chapter 22. Set

A *Set* is an unordered collection of zero or more elements with no key. Element equality is supported, and the values of the elements are relevant.

Only unique elements are supported. A request to add an element that already exists is ignored.

An example of a *Set* is a program that creates a packing list for a box of free samples to be sent to a warehouse customer. The program searches a database of in-stock merchandise, and selects ten items at random whose price is below a threshold level. Each item is then added to the *Set*. The *Set* does not allow an item to be added if it is already present in the collection, ensuring that a customer does not get two samples of a single product. The *Set* is not sorted, and elements of the *Set* cannot be located by key.

Figure 2 on page 19 gives an overview of the properties of a set and its relationship to other flat collections.

The *Set* also offers typical set functions such as union, intersection, and difference.

Derivation

Collection
Equality Collection
Set

Interface Name	Filestem
ISSet	sset

Members

All members of flat collections are described in Chapter 7, "Introduction to Flat Collections" on page 45. The following members are provided for set:

Method	Page	Method	Page
add	48	isEmpty	58
addAllFrom	48	isFull	59
addDifference	51	locate	59
addIntersection	51	locateOrAdd	62
addUnion	53	maxNumberOfElements	63
allElementsDo	53	newCursor	63
anyElement	54	newElementCursor	63
assign	54	notEqual	64
contains	55	numberOfElements	64
containsAllFrom	55	remove	65
copy	55	removeAll	66
Destructor	47	removeAt	66
differenceWith	56	replaceAt	68
elementAt	56	setToFirst	68
equal	58	setToNext	69
Initializer Method	47	unionWith	71
intersectionWith	58		
isBounded	58		

You can use an `ISElementCursor` with a `Set`. The members for `ISElementCursor` are described in Chapter 29, “Cursor” on page 115.

Required Operations

For `ISSet`, the operations listed below are required for the element type. You can either use the default operations from `ISOps` or override them with your own implementation.

Element Type

- `Assign()`
- `Compare()`
- `Equal()`

A coding example for a `Set` is provided in the appendix in “Coding Example for Set” on page 168.

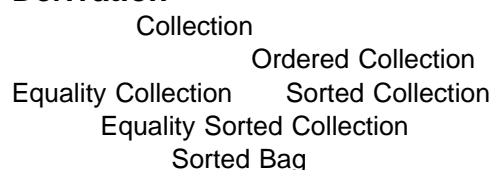
Chapter 23. Sorted Bag

A *Sorted Bag* is an ordered collection of zero or more elements with no key. Both element equality and multiple elements are supported.

An example of using a Sorted Bag is a program for entering observations on the types of stones found in a riverbed. Each time you find a stone on the riverbed, you enter the stone's mineral type into the collection. You can enter the same mineral type for several stones, because a Sorted Bag supports multiple elements. You can search for stones of a particular mineral type, and you can determine the number of observations of stones of that type. You can also display the contents of the collection, sorted by mineral type, if you want a complete list of observations made to date.

Figure 2 on page 19 gives an overview of the properties of a Sorted Bag and its relationship to other flat collections.

Derivation



Interface Name	Filestem
ISSortedBag	ssb

Members

All members of flat collections are described in Chapter 7, "Introduction to Flat Collections" on page 45. The following members are provided for Sorted Bag:

Method	Page	Method	Page
add	48	isFirst	59
addAllFrom	48	isFull	59
addDifference	51	isLast	59
addIntersection	51	lastElement	59
addUnion	53	locate	59
allElementsDo	53	locateNext	61
anyElement	54	locateOrAdd	62
assign	54	maxNumberOfElements	63
compare	54	newCursor	63
contains	55	newElementCursor	63
containsAllFrom	55	newOrderedCursor	64
copy	55	notEqual	64
Destructor	47	numberOfDifferentElements	64
differenceWith	56	numberOfElements	64
elementAt	56	numberOfOccurrences	64
elementAtPosition	56	position	65
equal	58	remove	65
firstElement	58	removeAll	66
Initializer Method	47	removeAllOccurrences	66
intersectionWith	58	removeAt	66
isBounded	58	removeAtPosition	67
isEmpty	58	removeFirst	67

Sorted Bag

Method	Page	Method	Page
removeLast	67	setToNextDifferentElement	69
replaceAt	68	setToPosition	70
setToFirst	68	setToPrevious	70
setToLast	69	unionWith	71
setToNext	69		

You can use an `ISOrderedCursor` with a `SortedBag`. The members for `ISOrderedCursor` are described in Chapter 29, “Cursor” on page 115.

Required Operations

For `ISSortedBag`, the operations listed below are required for the element type. You can either use the default operations from `ISOps` or override them with your own implementation.

Element Type

- `Assign()`
- `Compare()`
- `Equal()`

Chapter 24. Sorted Map

A *Sorted Map* is an ordered collection of zero or more elements that have a key. Element equality is supported and the values of the elements are relevant. Elements are sorted by the value of their keys.

Only elements with unique keys are supported. A request to add an element whose key already exists in another element of the collection causes an exception to be thrown. A request to add a duplicate element is ignored.

An example of using a Sorted Map is a program that matches the names of rivers and lakes to their coordinates on a topographical map. The river or lake name is the key. You cannot add a lake or river to the collection if it is already present in the collection. You can display a list of all lakes and rivers, sorted by their names, and you can locate a given lake or river by its key, to determine its coordinates.

Figure 2 on page 19 gives an overview of the properties of a Sorted Map and its relationship to other flat collections.

Derivation

```

Equality Key Collection      Equality Sorted Collection
      Equality Key Sorted Collection
              Sorted Map
  
```

The diagram does not show all bases of Sorted Map. See Figure 4 for a complete illustration.

Interface Name	Filestem
ISSortedMap	ssm

Members

All members of flat collections are described in Chapter 7, "Introduction to Flat Collections" on page 45. The following members are provided for Sorted Maps:

Method	Page	Method	Page
add	48	elementWithKey	57
addAllFrom	48	equal	58
addDifference	51	firstElement	58
addIntersection	51	Initializer Method	47
addOrReplaceElementWithKey	52	intersectionWith	58
addUnion	53	isBounded	58
allElementsDo	53	isEmpty	58
anyElement	54	isFirst	59
assign	54	isFull	59
compare	54	isLast	59
contains	55	key	59
containsAllFrom	55	lastElement	59
containsAllKeysFrom	55	locate	59
containsElementWithKey	55	locateElementWithKey	60
copy	55	locateNext	61
Destructor	47	locateNextElementWithKey	61
differenceWith	56	locateOrAdd	62
elementAt	56	locateOrAddElementWithKey	62
elementAtPosition	56	maxNumberOfElements	63

Sorted Map

Method	Page	Method	Page
newCursor	63	removeFirst	67
newElementCursor	63	removeLast	67
newOrderedCursor	64	replaceAt	68
notEqual	64	replaceElementWithKey	68
numberOfElements	64	setToFirst	68
position	65	setToLast	69
remove	65	setToNext	69
removeAll	66	setToPosition	70
removeAt	66	setToPrevious	70
removeAtPosition	67	unionWith	71
removeElementWithKey	67		

You can use an `ISOrderedCursor` with a `SortedMap`. The members for `ISOrderedCursor` are described in Chapter 29, “Cursor” on page 115.

Required Operations

For `ISSortedMap`, the operations listed below are required for the element type and key type. You can either use the default operations from `ISOps` or override them with your own implementation.

Element Type

- `Assign()`
- `Equal()`
- `Key()`

Key Type

- `KeyCompare`

Chapter 25. Sorted Relation

A *Sorted Relation* is an ordered collection of zero or more elements that have a key. The elements are sorted by the value of their key. Element equality is supported, and the values of the elements are relevant.

The keys of the elements are not unique. You can add an element whether or not there is already an element in the collection with the same key.

An example of using a Sorted Relation is a program used by telephone operators to provide directory assistance. The computerized directory is a Sorted Relation whose key is the name of the individual or business associated with a telephone number. When a caller requests the number of a given person or company, the operator enters the name of that person or company to access the phone number. The collection can have multiple identical keys, because two individuals or companies might have the same name. The collection is sorted alphabetically, because once a year it is used as the source material for a printed telephone directory.

Figure 2 on page 19 gives an overview of the properties of a Sorted Relation and its relationship to other flat collections.

Derivation

```

Equality Key Collection    Equality Collection
      Equality Key Sorted Collection
            Sorted Relation
  
```

The diagram does not show all bases of Sorted Relation. See Figure 4 on page 26 for a complete illustration.

Interface Name	Filestem
ISSortedRelation	ssr

Members

All members of flat collections are described in Chapter 7, "Introduction to Flat Collections" on page 45. The following members are provided for Sorted Relation:

Method	Page	Method	Page
add	48	differenceWith	56
addAllFrom	48	elementAt	56
addDifference	51	elementAtPosition	56
addIntersection	51	elementWithKey	57
addOrReplaceElementWithKey	52	equal	58
addUnion	53	firstElement	58
allElementsDo	53	Initializer Method	47
anyElement	54	intersectionWith	58
assign	54	isBounded	58
compare	54	isEmpty	58
contains	55	isFirst	59
containsAllFrom	55	isFull	59
containsAllKeysFrom	55	isLast	59
containsElementWithKey	55	key	59
copy	55	lastElement	59
Destructor	47	locate	59

Sorted Relation

Method	Page	Method	Page
locateElementWithKey	60	removeAllElementsWithKey	66
locateNext	61	removeAt	66
locateNextElementWithKey	61	removeAtPosition	67
locateOrAdd	62	removeElementWithKey	67
locateOrAddElementWithKey	62	removeFirst	67
maxNumberOfElements	63	removeLast	67
newCursor	63	replaceAt	68
newElementCursor	63	replaceElementWithKey	68
newOrderedCursor	64	setToFirst	68
notEqual	64	setToLast	69
numberOfDifferentKeys	64	setToNext	69
numberOfElements	64	setToNextWithDifferentKey	70
numberOfElementsWithKey	64	setToPosition	70
position	65	setToPrevious	70
remove	65	unionWith	71
removeAll	66		

You can use an `ISOrderedCursor` with a `SortedRelation`. The members for `ISOrderedCursor` are described in Chapter 29, “Cursor” on page 115.

Required Operations

For `ISSortedRelation`, the operations listed below are required for the element type and key type. You can either use the default operations from `ISOps` or override them with your own implementation.

Element Type

- `Assign()`
- `Equal()`
- `Key()`

Key Type

- `KeyCompare`

Chapter 26. Sorted Set

A *Sorted Set* is an ordered collection of zero or more elements with element equality but no key. Only unique elements are supported and a request to add an element that already exists is ignored. The value of the elements is relevant.

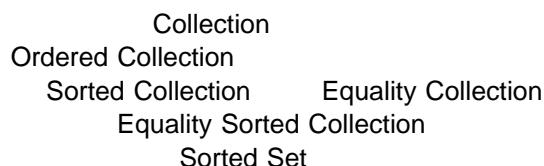
The elements of a Sorted Set are ordered such that the value of each element is less than or equal to the value of its successor.

The element with the smallest value currently in a Sorted Set is called the *first* element. The element with the largest value is called the *last* element. When an element is added, it is placed in the Sorted Set according to the defined ordering relation.

An example of using a Sorted Set is a program that tests numbers to see if they are prime. Two complementary Sorted Sets are used, one for prime numbers, and one for nonprime numbers. When you enter a number, the program first looks in the set of nonprime numbers. If the value is found there, the number is nonprime. If the value is not found there, the program looks in the set of prime numbers. If the value is found there, the number is prime. Otherwise the program determines whether the number is prime or nonprime, and places it in the appropriate Sorted Set. The program can also display a list of prime or nonprime numbers, beginning at the first prime or nonprime following a given value, because the numbers in a Sorted Set are sorted from smallest to largest.

Figure 2 on page 19 gives an overview of the properties of a sorted set and its relationship to other flat collections.

Derivation



Interface Name

Filestem

ISSortedSet

sss

Members

All members of flat collections are described in Chapter 7, "Introduction to Flat Collections" on page 45. The following members are provided for Sorted Sets:

Method	Page	Method	Page
add	48	containsAllFrom	55
addAllFrom	48	copy	55
addDifference	51	Destructor	47
addIntersection	51	differenceWith	56
addUnion	53	elementAt	56
allElementsDo	53	elementAtPosition	56
anyElement	54	equal	58
assign	54	firstElement	58
compare	54	Initializer Method	47
contains	55	intersectionWith	58

Sorted Set

Method	Page	Method	Page
isBounded	58	remove	65
isEmpty	58	removeAll	66
isFirst	59	removeAt	66
isFull	59	removeAtPosition	67
isLast	59	removeFirst	67
lastElement	59	removeLast	67
locate	59	replaceAt	68
locateNext	61	setToFirst	68
locateOrAdd	62	setToLast	69
maxNumberOfElements	63	setToNext	69
newCursor	63	setToPosition	70
newElementCursor	63	setToPrevious	70
newOrderedCursor	64	unionWith	71
notEqual	64		
position	65		

You can use an `ISOrderedCursor` with a `SortedSet`. The members for `ISOrderedCursor` are described in Chapter 29, “Cursor” on page 115.

Required Operations

For the default interface variant `ISSortedSet`, the operations listed below are required for the element type and key type. You can either use the default operations from `ISOps` or override them with your own implementation.

Element Type

- `Assign()`
- `Compare()`
- `Equal()`

A coding example for a *Sorted Set* is provided in the appendix in “Coding Example for Sorted Set” on page 175.

Chapter 27. Stack

A *Stack* is a sequence with restricted access, it is an ordered collection of elements with no key and no element equality. The elements are arranged so that each collection has a first and a last element, each element except the last has a next element, and each element but the first has a previous element. The type and value of the elements are irrelevant and have no effect on the behavior of the Stack.

Elements are added to and deleted from the *top* of the Stack. Consequently, the elements of a Stack are in reverse chronological order.

A Stack is characterized by a last-in, first-out (LIFO) behavior.

An example of using a Stack is a program that keeps track of daily tasks that you have begun to work on but that have been interrupted. When you are working on a task and something else comes up that is more urgent, you enter a description of the interrupted task and where you stopped it into your program, and the task is pushed onto the Stack. Whenever you complete a present task, you ask the program for the most recently saved task that was interrupted. This task is popped off the Stack, and you resume your work where you left off. When you attempt to pop an item off the Stack and no item is available, you have completed all your tasks and you can go home.

Derivation

```

Collection
  Ordered Collection
    Sequential Collection
      Sequence
        Stack
  
```

Note that Stack is based on sequence but is not actually derived from it or from the other classes shown above. See "Restricted Access" for further details.

Interface Name	Filestem
ISStack	sstk

Members

All members of flat collections are described in Chapter 7, "Introduction to Flat Collections" on page 45. The following members are provided for Stack:

Method	Page	Method	Page
add	48	firstElement	58
addAllFrom	48	Initializer Method	47
addAsLast	49	isBounded	58
allElementsDo	53	isEmpty	58
anyElement	54	isFirst	59
assign	54	isFull	59
compare	54	isLast	59
copy	55	lastElement	59
Destructor	47	numberOfElements	63
elementAt	56	newCursor	63
elementAtPosition	56	newElementCursor	63

Stack

Method	Page	Method	Page
newOrderedCursor	64	setToLast	69
numberOfElements	64	setToNext	69
pop	64	setToPosition	70
position	65	setToPrevious	70
push	65	top	71
removeAll	66		
removeLast	67		
setToFirst	68		

You can use an `ISOrderedCursor` with a `Stack`. The members for `ISOrderedCursor` are described in Chapter 29, “Cursor” on page 115.

Required Operations

For `ISStack`, the operations listed below are required for the element type. You can either use the default operations from `ISOps` or override them with your own implementation.

Element Type

- `Assign()`

A coding example for a `Stack` is provided in the appendix in “Coding Example for Stack” on page 186.

Reference : SOM Cross-language Collection Classes - Auxiliary Classes

Chapter 28. Global	113
Chapter 29. Cursor	115
Public Member Functions	115
Chapter 30. Applicator	119
Chapter 31. Comparator	121
Chapter 32. Predicate	123
Chapter 33. Operations	125

Chapter 28. Global

The module `SSGlobal` is intended to simplify the include mechanism for C and C++ application programmers. It is used to automatically include language usage bindings for applicators, cursors, comparators, predicates, and operations.

The include of `ssglobal.h` for a C application programmer should precede the first include statement for a specific collection.

IDL filestem

`ssglobal`

Chapter 29. Cursor

Each collection class defines its own cursor class. All of these cursor classes are derived from one of the following classes:

- ISElementCursor
- ISOrderedCursor

ISOrderedCursor is derived from ISElementCursor, and ISElementCursor is in turn derived from ISCursor. Only cursors of ordered collections are derived from ISOrderedCursor. Cursors from unordered collections are derived from ISElementCursor, and only know the member functions from ISElementCursor and ISCursor.

This chapter describes the general member functions of these three cursor classes as well as the specific member functions provided for specific collections. You can obtain cursor objects by using the collection member newCursor(), newElementCursor(), or newOrderedCursor(). The newCursor() member creates a cursor of the collection to which it is applied.

Each cursor object is associated with a collection object. A cursor function merely calls the corresponding function for this collection.

IDL filestem

- scursor
- secursor
- socursor

C and C++ application programmers include the appropriate ssglobal include file in order to receive the binding definitions for all cursors.

Members

The cursor classes define the following methods:

Method	Page	Method	Page
copy	116	equal	116
isValid	116	setToFirst	116
invalidate	116	setToLast	117
element	116	setToNext	117
notEqual	116	setToPrevious	117

Public Member Functions

Constructor

A cursor should be constructed only by calling the collection methods newCursor, newElementCursor, or newOrderedCursor.

copy

void **copy** (in ISCursor cursor) ;

Copies the given cursor to this cursor. This cursor now points to where the given cursor points.

Precondition: The given cursor and this cursor must refer to the same collection type.

Note: This precondition cannot be checked.

isValid

boolean **isValid** () ;

Returns 1 if the cursor points to an element of the associated collection.

invalidate

void **invalidate** () ;

Invalidates the cursor; that is, it no longer points to an element of the associated collection.

element

SOMObject **element** () ;

Returns a constant reference to the element of the associated collection to which the cursor points.

Precondition: The cursor must point to an element of the associated collection.

Exception: ICursorInvalidException

notEqual

boolean **notEqual** (in ISCursor cursor) ;

Returns 1 if the cursor does not point to the same element (of the same collection) as the given cursor.

equal

boolean **equal** (in ISCursor cursor) ;

Returns 1 if the cursor points to the same element (of the same collection) as the given cursor.

setToFirst

boolean **setToFirst** () ;

Sets the cursor to the first element of the associated collection in iteration order. Invalidates the cursor if the collection is empty (if no first element exists).

Return Value: Returns 1 if the associated collection is not empty.

setToLast

boolean **setToLast** () ;

Sets the cursor to the last element of the associated collection in iteration order. Invalidates the cursor if the collection is empty (no last element exists); this function is only available for cursors of ordered collections.

Return Value Returns 1 if the associated collection was not empty.

setToNext

boolean **setToNext** () ;

Sets the cursor to the next element in the associated collection in iteration order, and the cursor is invalidated if there are no more elements left to be visited.

Return Value Returns 1 if there was a next element.

Precondition: The cursor must point to an element of the associated collection.

Exception: `ICursorInvalidException`

setToPrevious

boolean **setToPrevious** () ;

Sets the cursor to the previous element of the associated collection in iteration order and it invalidates the cursor if no such element exists. This function is only available for cursors of ordered collections.

Return Value: Returns 1 if a previous element exists.

Precondition: The cursor must point to an element of the associated collection.

Exception: `ICursorInvalidException`

Chapter 30. Applicator

The interface ISApplicator acts as an abstract class. You must not create an instance of this class.

You must inherit from this class and create an appropriate instance. Whenever a collection method requires an ISApplicator argument the created instance must be used in the collection's method call.

IDL filestem

sappl

C and C++ application programmers include the appropriate `ssglobal` include file in order to receive the binding definitions for the applicator.

Members

The applicator class defines the following methods:

applyTo

boolean **applyTo** (in SOMObject element) ;

The user must override this method while subclassing from this interface.

The application programmer defines the return value when overwriting the **applyTo** method in the derived class.

Exception

- IApplicatorOverrideException
- IUserApplicatorException

Example

```
/* aSet is an instance of any Set variant... */
/* SetApplicator is derived from ISApplicator */
/*   with an overridden applyTo() method... */

SetApplicator applicator;
...
applicator = _somNew(_SetApplicator);
...
_allElementsDo(aSet,ev,applicator);
...
```


Chapter 31. Comparator

The interface `ISComparator` acts as an abstract class. You must inherit from this class whenever a collection method requires an `ISComparator` argument.

IDL filestem

scomp

C and C++ application programmers include the appropriate `ssglobal` include file in order to receive the binding definitions for the comparator.

Members

The comparator class defines the following methods:

compare

boolean **compare** (in `SOMObject` element) ;

The user must override this method while subclassing from this interface.

The application programmer defines the return value when overwriting the **applyTo** method in the derived class.

Exception

- `ISComparatorOverrideException`
- `IUserComparatorException`

Example

```
/* aSet is an instance of any Set variant... */
/* SetComparator is derived from ISComparator */
/* with an overridden compare method          */

SetComparator comparator;
...
comparator = _somNew(_SetComparator);
...
ISASet_sort(aSet,ev,comparator);
...
```


Chapter 32. Predicate

The interface `ISPredicate` acts as an abstract class. You must not create an instance of this class.

You must inherit from this class and create an appropriate instance. Whenever a collection method requires an `ISPredicate` argument the created instance must be used in the collection's method call.

IDL filestem

spred

C and C++ application programmers include the appropriate `ssglobal` include file in order to receive the binding definitions for the predicate.

Members

The predicate class defines the following methods:

evaluateFor

`boolean evaluateFor (in SOMObject element) ;`

The user must override this method while subclassing from this interface.

The application programmer defines the return value when overwriting the **evaluateFor** method in the derived class.

Exception

- `IPredicateOverrideException`
- `IUserPredicateException`

Example

```
/* IDL:                                     */

interface SetPredicate : ISPredicate {

...
evaluateFor : override;
...
}

/* aSet is an instance of any Set variant */
/* SetPredicate is derived from ISPredicate */
/* with an overridden evaluateFor() method */
...
SetPredicate predicate;
...
predicate = _somNew(_SetPredicate);
...
_removeAllWithPredicate(aSet,ev,predicate);
...
```


Chapter 33. Operations

The operations class `ISOps` represents the generic class for operations on collectible elements and keys, and acts as an abstract class. You must not create an instance of this class.

You must inherit from this interface and create a appropriate instance. An instance of an `ISOps` derived interface must be specified within the initializer method of a specific collection.

You should override several of below methods which are called internally from the SOM Collection Classes implementation.

You must not destroy any of the created operations instances. The fate of an operations instance is the responsibility of the collection which is using it.

Although you must not specify an operations instance in multiple collections, internally it can occur that several collections use the same operations instance; this happens when a collection is used to construct another collection. The operations object disappears when the last using collection is deleted.

IDL filestem

ssops

C and C++ application programmers include the appropriate `ssglobal` include file in order to receive the binding definitions for operations.

Members

The operations class define the following methods:

Assign

```
void Assign ( inout SOMObject e1, in SOMObject e2 ) ;
```

You may override this method in order to define a different implementation. In the normal case the default behavior is sufficient for standard usage.

Compare

```
long Compare ( in SOMObject e1, in SOMObject e2 ) ;
```

The default implementation of the provided `Compare()` operation is a comparison of pointers to `e1` and `e2`. You should override this method for non key collections in order to define the ordering relation between elements. Usually this method calls other methods defined for the element `SOMObject`.

Equal

```
boolean Equal ( in SOMObject e1, in SOMObject e2 ) ;
```

The default implementation of the provided `Equal()` operation is a comparison of pointers to `e1` and `e2`. You should override this method for non key collections like maps and relations in order to define the equality relation between elements. Usually this method calls other methods defined for the element `SOMObject`.

Hash

long **Hash** (in SOMObject e1, in unsigned long value) ;

The default implementation of the provided Hash() always returns the value -1. You should override this method in order to define a hash function for a non key collection implementation variant based on a hash function. Usually this method calls other methods defined for the element SOMObject.

Key

SOMObject **Key** (in SOMObject element) ;

This method is required for key collections only. The default Key operation always returns the element. You should override this method in order to extract a key object from stored element within a collection. Usually this method calls another method defined for element SOMObject.

KeyCompare

long **KeyCompare** (in SOMObject k1, in SOMObject k2) ;

This method is required for key collections only. The default implementation of the provided KeyCompare() operation is a comparison of pointers to k1 and k2. You should override this method in order to define the ordering relation for keys. Usually this method calls other methods defined for the key SOMObject.

KeyEqual

boolean **KeyEqual** (in SOMObject k1, in SOMObject k2) ;

This method is required for key collections Key Bag and Relation. The default implementation of the provided KeyEqual() operation is a comparison of pointers to k1 and k2. You should override this method in order to define the equality relation. Usually this method calls other methods defined for the key SOMObject.

KeyHash

long **KeyHash** (in SOMObject key, in unsigned long value) ;

The default implementation of the provided KeyHash() always returns the value -1. This method is required for key collections only. You should override this method in order to define a hash function for a collection implementation variant based on a hash function. Usually this method calls other methods defined for the key SOMObject.

Reference : SOM Cross-language Collection Classes - Abstract Classes

Chapter 34. Collection	129
Chapter 35. Equality Collection	131
Chapter 36. Key Collection	133
Chapter 37. Ordered Collection	135
Chapter 38. Sorted Collection	137
Chapter 39. Sequential Collection	139
Chapter 40. Equality Key Collection	141
Chapter 41. Key Sorted Collection	143
Chapter 42. Equality Sorted Collection	145
Chapter 43. Equality Key Sorted Collection	147

Chapter 34. Collection

Derivation

Collection does not have any bases. Because collection is an abstract class, it cannot be used to create any objects. The following abstract classes are derived from collection:

- Key collection
- Equality collection
- Ordered collection

The concrete class heap is defined by collection.

Figure 4 on page 26 shows the relationship of collection to the class hierarchy.

IDL Stem	Interface Name
sacclct	ISACollection

Members

The following methods are provided for Collection:

Method	Page	Method	Page
add	48	maxNumberOfElements	63
addAllFrom	48	newCursor	63
anyElement	54	newElementCursor	63
copy	54	numberOfElements	64
elementAt	56	removeAll	66
elementAtPosition	56	removeAt	66
isBounded	58	replaceAt	68
isEmpty	58	setToFirst	68
isFull	59	setToNext	69

Chapter 35. Equality Collection

Because *equality collection* acts as an abstract class, it must not be used to create any objects. The equality collection defines the interfaces for the property of element equality.

Derivation

Collection

Equality Collection

The following abstract classes are derived from equality collection:

- Equality key collection
- Equality sorted collection

The following concrete classes are defined by equality collection:

- Set
- Bag
- Equality Sequence

Figure 4 on page 26 shows the relationship of equality collection to the class hierarchy.

IDL Stem	Interface Name
saequal	ISAEqualityCollection

Members

The equality collection class defines the following member functions, described in Chapter 7, "Introduction to Flat Collections" on page 45:

Method	Page	Method	Page
contains	55	locateOrAdd	62
containsAllFrom	55	numberOfOccurrences	64
locate	59	remove	65
locateNext	61	removeAllOccurrences	66

Chapter 36. Key Collection

Because *key collection* acts as an abstract class, it must not be used to create any objects. The key collection inherits from collection and defines the interfaces for the key property.

Derivation

Collection
Key Collection

The following abstract classes are derived from key collection:

- Equality key collection
- Key sorted collection

The following concrete classes are defined by key collection:

- Key set
- Key bag

Figure 4 on page 26 shows the relationship of key collection to the class hierarchy.

IDL Stem	Interface Name
sakey	ISAKeyCollection

Members

The key collection class defines the following member functions, described in Chapter 7, "Introduction to Flat Collections" on page 45:

Method	Page	Method	Page
addOrReplaceElementWithKey	52	locateOrAddElementWithKey	62
containsAllKeysFrom	55	numberOfDifferentKeys	64
containsElementWithKey	55	numberOfElementsWithKey	64
elementWithKey	57	removeAllElementsWithKey	66
key	59	removeElementWithKey	67
locateElementWithKey	60	replaceElementWithKey	68
locateNextElementWithKey	61	setToNextWithDifferentKey	70

Chapter 37. Ordered Collection

Because *ordered collection* acts as an abstract class, it must not be used to create any objects. The ordered collection defines the interfaces for the property of ordered elements.

Derivation

Collection

Ordered Collection

The following abstract classes are derived from ordered collection:

- Sorted collection
- Sequential collection

Figure 4 on page 26 shows the relationship of ordered collection to the class hierarchy.

IDL Stem

Interface Name

saorder

ISAOorderedCollection

Members

The ordered collection class defines the following member functions, described in Chapter 7, "Introduction to Flat Collections" on page 45:

Method	Page	Method	Page
elementAtPosition	56	removeAtPosition	67
firstElement	58	removeFirst	67
isFirst	59	removeLast	67
isLast	59	setToLast	69
lastElement	59	setToPosition	70
newOrderedCursor	64	setToPrevious	70
positionAt	65		

Chapter 38. Sorted Collection

Because *sorted collection* acts as an abstract class, it must not be used to create any objects. The sorted collection inherits from ordered collection and defines the interfaces for the properties of sorted elements.

Derivation

Collection
 Ordered Collection
 Sorted Collection

The following abstract classes are derived from sorted collection:

- Equality sorted collection
- Key sorted collection

Figure 4 on page 26 shows the relationship of sorted collection to the class hierarchy.

IDL Stem	Interface Name
sasrt	ISASortedCollection

Members

The sorted collection class inherits all its members from its bases.

Chapter 39. Sequential Collection

Because *sequential collection* acts as an abstract class, it must not be used to create any objects. The sequential collection inherits from ordered collection and defines the interfaces for the properties of ordered elements.

Derivation

```
Collection
  Ordered Collection
    Sequential Collection
```

The following concrete classes are defined by sequential collection:

- Sequence
- Equality sequence

Figure 4 on page 26 shows the relationship of sequential collection to the class hierarchy.

IDL Stem	Interface Name
sasqntl	ISASequentialCollection

Members

Sequential collection defines the following member functions:

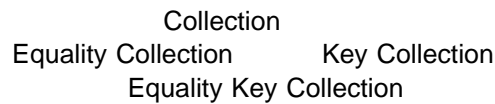
Method	Page	Method	Page
addAsFirst	49	addAtPosition	50
addAsLast	49	position	65
addAsNext	50	sort	71
addAsPrevious	50		

Chapter 40. Equality Key Collection

Because *equality key collection* acts as an abstract class, it must not be used to create any objects. It defines the interfaces for the following properties:

- Element equality
- Key equality

Derivation



Equality key sorted collection is an abstract class that is derived from equality key collection. The following concrete classes are defined by equality key collection:

- Map
- Relation

Figure 4 on page 26 shows the relationship of equality key collection to the whole class hierarchy.

IDL Stem	Interface Name
saeqkey	ISAEqualityCollection

Members

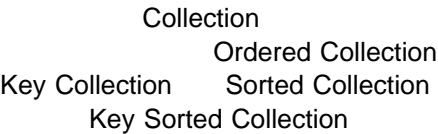
All the members of equality key sorted collection are inherited from its base classes.

Chapter 41. Key Sorted Collection

Because *key sorted collection* acts as an abstract class, it must not be used to create any objects. The key sorted collection inherits from sorted collection and key collection. It defines the interfaces for the following properties:

- Key equality
- Sorted elements

Derivation



The equality key sorted collection is an abstract class that is derived from key sorted collection. The following concrete classes are defined by key sorted collection:

- Key sorted set
- Key sorted bag

Figure 4 on page 26 shows the relationship of key sorted collection to the class hierarchy.

IDL Stem	Interface Name
saksort	ISASortedCollection

Members

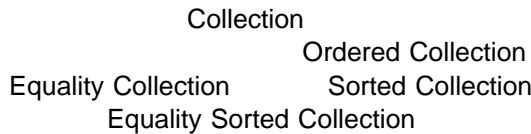
The key sorted collection class inherits all member functions from its base classes.

Chapter 42. Equality Sorted Collection

Because *equality sorted collection* acts as an abstract class, it must not be used to create any objects. It defines the interfaces for the following properties:

- Element equality
- Sorted elements

Derivation



Equality key sorted collection is an abstract class that is derived from equality sorted collection. The following concrete classes are defined by equality sorted collection:

- Sorted set
- Sorted bag

Figure 4 on page 26 shows the relationship of equality sorted collection to the class hierarchy.

IDL Stem	Interface Name
saeqsrt	ISAEqualitySortedCollection

Members

All members of equality sorted collection are inherited from its base classes.

Chapter 43. Equality Key Sorted Collection

Equality key sorted collection is an abstract class that defines the interfaces for the following properties:

- Element equality
- Key equality
- Sorted elements

Because *equality key sorted collection* acts as an abstract class, it must not be used to create any objects.

Derivation

Equality key sorted collection is derived from the following three abstract classes:

- Key sorted collection
- Equality sorted collection
- Equality key sorted collection

For information on the bases of these classes, see Figure 4 on page 26

The following concrete classes are defined by equality key sorted collection:

- Sorted map
- Sorted relation

Figure 4 on page 26 shows the relationship of equality key sorted collection to the class hierarchy.

IDL Stem	Interface Name
saeksrt	ISAEqualityKeySortedCollection

Members

All the members of equality key sorted collection are inherited from its base classes.

Appendix A. Coding Samples: Source Code and Header Files

Coding Example for Deque

The following program uses the default deque class, `ISDeque`, to create a deque. It fills the deque with characters by adding them to the back end. The program then removes the characters from alternating ends of the deque (beginning with the front end) until the deque is empty.

The program uses the applicator interface, `ISApplicator`, when printing the collection. It uses the `addAsLast()` function to fill the deque and the `numberOfElements()` function to determine the deque's size. It uses the functions `firstElement()`, `removeFirst()`, `lastElement()`, and `removeLast()` to empty the deque.

```
----- exp3ele.idl -----
#ifndef CHAROBJECT_IDL
#define CHAROBJECT_IDL

#include <somobj.idl>

interface CharObject :SOMObject
{

attribute   char charValue;

void        CharObject_withChar(inout somInitCtrl ctrl,
                                in char aChar) ;
void        print_charValue() ;

#if defined __SOMIDL__
implementation
{
releaseorder : CharObject_withChar, _get_charValue,
               _set_charValue, print_charValue ;

CharObject_withChar      : init ;
somDefaultInit           : override ;
somDestruct               : override ;

dllname = "letterdq.dll" ;

passthru C_h_before = "#include <string.h>" ;

};
#endif // __SOMIDL__

};

#endif // CHAROBJECT_IDL

----- exp3ops.idl -----
#ifndef CHAROBJECTSOPS_IDL
#define CHAROBJECTSOPS_IDL

#include <ssops.idl>

interface CharObjectsOps : ISOps
{
```

```

#if defined __SOMIDL__
implementation
{
Assign          : override ;
Compare         : override ;
Equal           : override ;

somDefaultInit  : override ;
somDestruct     : override ;

dllname = "exp3.dll" ;

passthru C_h_before = "#include <charObj.h>";
};

#endif // __SOMIDL__

};
#endif // CHAROBJECTSOPS_IDL

----- exp3appl.idl -----
#ifndef PRAPPL_IDL
#define PRAPPL_IDL

#include <sappl.idl>

interface PrintAppl : ISApplicator
{

#if defined __SOMIDL__
implementation
{

somDefaultInit      : override ;
somDestruct         : override ;
applyTo             : override ;

dllname= "exp3.dll" ;

passthru C_h_before = "#include <ssglobal.h>"
                      "#include <exp3ele.h>";

};
#endif // __SOMIDL__

};

#endif // PRAPPL_IDL

----- exp3ele.c -----
/*
 * This file was generated by the SOM Compiler and Emitter Framework.
 * Generated using:
 *     SOM Emitter emitctm.dll: 2.41
 */

#ifndef SOM_Module_charobj_Source
#define SOM_Module_charobj_Source
#endif
#define CharObject_Class_Source

#ifdef _MVS
#include <DD:IH(exp3ele)>
#else

```

```

#include "exp3ele.ih"
#endif

SOM_Scope void SOMLINK CharObject_withChar(CharObject somSelf,
                                           Environment *ev,
                                           somInitCtrl* ctrl,
                                           char aChar)
{
    CharObjectData *somThis; /* set in BeginInitializer */
    somInitCtrl globalCtrl;
    somBooleanVector myMask;
    CharObjectMethodDebug("CharObject","CharObject_withChar");
    CharObject_BeginInitializer_CharObject_withChar;

    CharObject_Init_SOMObject_somDefaultInit(somSelf, ctrl);

    somThis->charValue = aChar ;
}

SOM_Scope void SOMLINK print_charValue(CharObject somSelf, Environment *ev)
{
    CharObjectData *somThis = CharObjectGetData(somSelf);
    CharObjectMethodDebug("CharObject","print_charValue");

    somPrintf("%s" , & (somThis->charValue)) ;
}

SOM_Scope void SOMLINK somDefaultInit(CharObject somSelf, somInitCtrl* ctrl)
{
    CharObjectData *somThis; /* set in BeginInitializer */
    somInitCtrl globalCtrl;
    somBooleanVector myMask;
    CharObjectMethodDebug("CharObject","somDefaultInit");
    CharObject_BeginInitializer_somDefaultInit;

    CharObject_Init_SOMObject_somDefaultInit(somSelf, ctrl);

    /*
    Just overridden with default for performance reasons
    */
}

SOM_Scope void SOMLINK somDestruct(CharObject somSelf, octet doFree,
                                   somDestructCtrl* ctrl)
{
    CharObjectData *somThis; /* set in BeginDestructor */
    somDestructCtrl globalCtrl;
    somBooleanVector myMask;
    CharObjectMethodDebug("CharObject","somDestruct");
    CharObject_BeginDestructor;

    /*
    Just overridden with default for performance reasons
    */

    CharObject_EndDestructor;
}

----- exp3ops.c -----

/*
 * This file was generated by the SOM Compiler.
 * Generated using:
 *     SOM incremental update: 2.41
 */

/*
 * This file was generated by the SOM Compiler and Emitter Framework.
 * Generated using:

```

```

*      SOM Emitter emitctm.dll: 2.41
*/

#ifndef SOM_Module_charobjsops_Source
#define SOM_Module_charobjsops_Source
#endif
#define CharObjectsOps_Class_Source

#ifdef _MVS
#include <DD:IH(exp3ops)>
#else
#include "exp3ops.ih"
#endif

SOM_Scope void SOMLINK Assign(CharObjectsOps somSelf, Environment *ev,
                              SOMObject e1, SOMObject e2)
{
    /* CharObjectsOpsData *somThis = CharObjectsOpsGetData(somSelf); */

    char val2 ;

    CharObjectsOpsMethodDebug("CharObjectsOps","Assign");

    val2 = __get_charValue(e2, ev) ;
    __set_charValue(e1, ev, val2) ;
}

SOM_Scope long SOMLINK Compare(CharObjectsOps somSelf, Environment *ev,
                               SOMObject e1, SOMObject e2)
{
    /* CharObjectsOpsData *somThis = CharObjectsOpsGetData(somSelf); */

    char val1 ;
    char val2 ;

    CharObjectsOpsMethodDebug("CharObjectsOps","Compare");

    val1 = __get_charValue(e1, ev) ;
    val2 = __get_charValue(e2, ev) ;
    return (val1 - val2) ;
}

SOM_Scope boolean SOMLINK Equal(CharObjectsOps somSelf, Environment *ev,
                                 SOMObject e1, SOMObject e2)
{
    /* CharObjectsOpsData *somThis = CharObjectsOpsGetData(somSelf); */

    char val1 ;
    char val2 ;

    CharObjectsOpsMethodDebug("CharObjectsOps","Equal");

    val1 = __get_charValue(e1, ev) ;
    val2 = __get_charValue(e2, ev) ;
    return (val1 == val2) ;
}

SOM_Scope void SOMLINK somDefaultInit(CharObjectsOps somSelf,
                                       somInitCtrl* ctrl)
{
    CharObjectsOpsData *somThis; /* set in BeginInitializer */
    somInitCtrl globalCtrl;
    somBooleanVector myMask;
    CharObjectsOpsMethodDebug("CharObjectsOps","somDefaultInit");
    CharObjectsOps_BeginInitializer_somDefaultInit;

    CharObjectsOps_Init_ISOps_somDefaultInit(somSelf, ctrl);
}

```



```

        /*
        Just overridden with default for performance reasons
        */
    }

SOM_Scope void SOMLINK somDestruct(CharObjectsOps somSelf, octet doFree,
                                   somDestructCtrl* ctrl)
{
    CharObjectsOpsData *somThis; /* set in BeginDestructor */
    somDestructCtrl globalCtrl;
    somBooleanVector myMask;
    CharObjectsOpsMethodDebug("CharObjectsOps","somDestruct");
    CharObjectsOps_BeginDestructor;

    /*
    Just overridden with default for performance reasons
    */

    CharObjectsOps_EndDestructor;
}

----- exp3appl.c -----
/*
 * This file was generated by the SOM Compiler and Emitter Framework.
 * Generated using:
 *     SOM Emitter emitctm.dll: 2.41
 */

#ifndef SOM_Module_prappl_Source
#define SOM_Module_prappl_Source
#endif
#define PrintAppl_Class_Source

#ifdef _MVS
#include <DD:IH(exp3appl)>
#else
#include "exp3appl.i.h"
#endif

SOM_Scope void SOMLINK somDefaultInit(PrintAppl somSelf, somInitCtrl* ctrl)
{
    PrintApplData *somThis; /* set in BeginInitializer */
    somInitCtrl globalCtrl;
    somBooleanVector myMask;
    PrintApplMethodDebug("PrintAppl","somDefaultInit");
    PrintAppl_BeginInitializer_somDefaultInit;

    PrintAppl_Init_ISApplicator_somDefaultInit(somSelf, ctrl);

    /*
    Just overridden for performance reasons
    */
}

SOM_Scope void SOMLINK somDestruct(PrintAppl somSelf, octet doFree,
                                   somDestructCtrl* ctrl)
{
    PrintApplData *somThis; /* set in BeginDestructor */
    somDestructCtrl globalCtrl;
    somBooleanVector myMask;
    PrintApplMethodDebug("PrintAppl","somDestruct");
    PrintAppl_BeginDestructor;

    /*
    Just overridden for performance reasons
    */
}

```

```

        PrintAppl_EndDestructor;
    }

SOM_Scope boolean SOMLINK applyTo(PrintAppl somSelf, Environment *ev,
                                   SOMObject element)
{
    /* PrintApplData *somThis = PrintApplGetData(somSelf); */
    PrintApplMethodDebug("PrintAppl","applyTo");

    CharObject_print_charValue(element , ev ) ;

    return (TRUE) ;

}

----- exp3ini.c -----
#include <exp3ele.h>
#include <exp3ops.h>
#include <exp3appl.h>

#ifdef __IBMC__
#pragma linkage(SOMInitModule, system)
#endif

SOMEXTERN void SOMLINK SOMInitModule (long majorVersion,
                                       long minorVersion,
                                       string className)
{
    CharObjectNewClass(CharObject_MajorVersion , CharObject_MinorVersion);
    CharObjectsOpsNewClass(CharObjectsOps_MajorVersion , CharObjectsOps_MinorVersion);
    PrintApplNewClass(PrintAppl_MajorVersion , PrintAppl_MinorVersion);
    return;
}

----- exp3main.c -----

/*
 * This file was generated by the SOM Compiler and Emitter Framework.
 * Generated using:
 *     SOM Emitter emitctm.dll: 2.41
 */

#ifndef SOM_Module_charobjsops_Source
#define SOM_Module_charobjsops_Source
#endif
#define CharObjectsOps_Class_Source

#ifdef _MVS
#include <DD:IH(exp3ops)>
#else
#include "exp3ops.ih"
#endif

SOM_Scope long SOMLINK Compare(CharObjectsOps somSelf,
                               Environment *ev,
                               SOMObject e1, SOMObject e2)
{
    /* CharObjectsOpsData *somThis = CharObjectsOpsGetData(somSelf); */

    char val1 ;
    char val2 ;

    CharObjectsOpsMethodDebug("CharObjectsOps","Compare");

    val1 = __get_charValue(e1, ev) ;
    val2 = __get_charValue(e2, ev) ;
    return (val1 - val2 ) ;

}

```

```

SOM_Scope boolean SOMLINK Equal(CharObjectsOps somSelf,
                                Environment *ev,
                                SOMObject e1, SOMObject e2)
{
    /* CharObjectsOpsData *somThis = CharObjectsOpsGetData(somSelf); */

    char val1 ;
    char val2 ;

    CharObjectsOpsMethodDebug("CharObjectsOps","Equal");

    val1 = __get_charValue(e1, ev) ;
    val2 = __get_charValue(e2, ev) ;
    return (val1 == val2) ;
}

SOM_Scope void SOMLINK somDefaultInit(CharObjectsOps somSelf,
                                       somInitCtrl* ctrl)
{
    CharObjectsOpsData *somThis; /* set in BeginInitializer */
    somInitCtrl globalCtrl;
    somBooleanVector myMask;
    CharObjectsOpsMethodDebug("CharObjectsOps","somDefaultInit");
    CharObjectsOps_BeginInitializer_somDefaultInit;

    CharObjectsOps_Init_ISOps_somDefaultInit(somSelf, ctrl);

    /*
    Just overridden with default for performance reasons
    */
}

SOM_Scope void SOMLINK somDestruct(CharObjectsOps somSelf,
                                    octet doFree,
                                    somDestructCtrl* ctrl)
{
    CharObjectsOpsData *somThis; /* set in BeginDestructor */
    somDestructCtrl globalCtrl;
    somBooleanVector myMask;
    CharObjectsOpsMethodDebug("CharObjectsOps","somDestruct");
    CharObjectsOps_BeginDestructor;

    /*
    Just overridden with default for performance reasons
    */

    CharObjectsOps_EndDestructor;
}

```

This program produces the following output:

```

Letterdq sample running ...
Adding characters to the back end of the queue : ...
Added to LetterDeque: T
Added to LetterDeque: e
Added to LetterDeque: q
Added to LetterDeque: i
Added to LetterDeque: k
Added to LetterDeque: b
Added to LetterDeque: o
Added to LetterDeque: n

```

```

Added to LetterDeque: f
Added to LetterDeque: x
Added to LetterDeque: j
Added to LetterDeque: m
Added to LetterDeque: s
Added to LetterDeque: o
Added to LetterDeque: e
Added to LetterDeque:
Added to LetterDeque:
Added to LetterDeque: a
Added to LetterDeque: y
Added to LetterDeque: d
Added to LetterDeque: g
Added to LetterDeque: .
Added to LetterDeque: o
Added to LetterDeque:
Added to LetterDeque: z
Added to LetterDeque: l
Added to LetterDeque: a
Added to LetterDeque: r
Added to LetterDeque: v
Added to LetterDeque:
Added to LetterDeque: p
Added to LetterDeque: u
Added to LetterDeque:
Added to LetterDeque: o
Added to LetterDeque:
Added to LetterDeque: w
Added to LetterDeque: r
Added to LetterDeque:
Added to LetterDeque: c
Added to LetterDeque: u
Added to LetterDeque:
Added to LetterDeque: h
Current number of elements in LetterDeque: 42
Content of LetterDeque:
Teqikbonfxjmsoe aydg.o zlarv pu o wr cu h
Reading from LetterDeque one element from front, one from back, and so on:
The quick brown fox jumps over a lazy dog.

```

Coding Example for Key Bag

The following program uses the default key bag class, `ISKeyBag`, to create a key bag for storing observations made on animals. The key of the class is the name of the animal. The program produces various reports regarding the observations. Then it removes all the extinct animals, which are stored in a sequence, from the key bag.

The program uses the `add()` function to fill the key bag and a `Cursor` to display the observations. It uses the following functions to produce the reports:

- `numberOfElements()`

- numberOfDifferentKeys()
- numberOfElementsWithKey()
- locateElementWithKey()
- setToNextElementWithKey()
- removeAllElementsWithKey()

```

----- explele.idl -----
#ifndef ANIMAL_IDL
#define ANIMAL_IDL

#include <somobj.idl>

interface AnimalsKey ;

interface Animal : SOMObject
{

void Animal_withNameAndProperty (inout somInitCtrl ctrl , in string aName , in
string aProperty) ;

string      get_Name() ;
string      get_Property() ;
AnimalsKey  get_Key() ;
boolean      equalAnimals(in Animal anotherAnimal) ;
void        printAnimal() ;

#if defined __SOMIDL__
implementation
{

releaseorder      : Animal_withNameAndProperty ,
                   get_Name , get_Property, get_Key,
                   equalAnimals,
                   printAnimal ;

Animal_withNameAndProperty : init ;
somDefaultInit             : override ;
somDestruct                : override ;

AnimalsKey  nameKey ;
string      property ;

dllname = "animal.dll" ;

passthru C_h_before = "#include <explkey.h>" \
                      "#include <string.h>" ;

};
#endif // __SOMIDL__

};
#endif // ANIMAL_IDL

----- explkey.idl -----
#ifndef ANIMALSKEY_IDL
#define ANIMALSKEY_IDL

#include <somobj.idl>

interface AnimalsKey : SOMObject
{

void AnimalsKey_withName (inout somInitCtrl ctrl , in string aName ) ;

string  get_Name() ;

```

```

#if defined __SOMIDL__
implementation
{

    releaseorder : AnimalsKey_withName , get_Name ;

    AnimalsKey_withName : init ;
    somDefaultInit      : override ;
    somDestruct         : override ;

    string  name ;

    dllname = "expl.dll" ;

    passthru C_h_before = "#include <string.h>" ;
};
#endif // __SOMIDL__

};
#endif // ANIMALSKEY_IDL

----- explops.idl -----
#ifndef ANIMALSOPS_IDL
#define ANIMALSOPS_IDL

#include <ssops.idl>

interface AnimalsOps : ISOps
{

    #if defined __SOMIDL__
    implementation
    {
        Key          : override ;
        KeyHash      : override ;
        KeyEqual     : override ;

        somDefaultInit : override ;
        somDestruct    : override ;

        string  name ;

        dllname = "expl.dll" ;

        passthru C_h_before = "#include <ssglobal.h>"
                               "#include <explele.h>"
                               "#include <explkey.h>" ;
    };
    #endif // __SOMIDL__

};
#endif // ANIMALSOPS_IDL

----- explele.c -----

/*
 * This file was generated by the SOM Compiler.
 * Generated using:
 *     SOM incremental update: 2.41
 */

/*
 * This file was generated by the SOM Compiler and Emitter Framework.
 * Generated using:
 *     SOM Emitter emitctm.dll: 2.41
 */

```

```

#ifndef SOM_Module_animal_Source
#define SOM_Module_animal_Source
#endif
#define Animal_Class_Source

#ifdef _MVS
#include <DD:IH(explele)>
#else
#include "explele.iH"
#endif

SOM_Scope void SOMLINK Animal_withNameAndProperty(Animal somSelf,
                                                    Environment *ev,
                                                    somInitCtrl* ctrl,
                                                    string aName,
                                                    string aProperty)
{
    AnimalData *somThis; /* set in BeginInitializer */
    somInitCtrl globalCtrl;
    somBooleanVector myMask;
    string ptr ;

    AnimalMethodDebug("Animal","Animal_withNameAndProperty");
    Animal_BeginInitializer_Animal_withNameAndProperty;

    Animal_Init_SOMObject_somDefaultInit(somSelf, ctrl);

    somThis->nameKey =
        (AnimalsKey) AnimalsKeyNew_AnimalsKey_withName(ev, aName);

    ptr = (string) SOMMalloc(strlen(aProperty) +1 ) ;
    strcpy (ptr, aProperty) ;
    somThis->property = ptr ;
}

SOM_Scope string SOMLINK get_Name(Animal somSelf, Environment *ev)
{
    AnimalData *somThis = AnimalGetData(somSelf);
    string retVal ;

    AnimalMethodDebug("Animal","get_Name");

    retVal = (string) AnimalsKey_get_Name(somThis->nameKey,ev);
    return (retVal) ;
}

SOM_Scope string SOMLINK get_Property(Animal somSelf, Environment *ev)
{
    AnimalData *somThis = AnimalGetData(somSelf);
    AnimalMethodDebug("Animal","get_Property");
    return (somThis->property);
}

SOM_Scope AnimalsKey SOMLINK get_Key(Animal somSelf, Environment *ev)
{
    AnimalData *somThis = AnimalGetData(somSelf);
    AnimalMethodDebug("Animal","get_Key");
    return (somThis->nameKey);
}

SOM_Scope boolean SOMLINK equalAnimals(Animal somSelf, Environment *ev,
                                         Animal anotherAnimal)
{
    AnimalData *somThis = AnimalGetData(somSelf);
    boolean retVal = FALSE ;
    string ptr1 ;
    string ptr2 ;
    string ptr3 ;
    string ptr4 ;
    AnimalMethodDebug("Animal","equalAnimals");

```

```

    ptr1 = (string) Animal_get_Name (anotherAnimal , ev) ;
    ptr2 = (string) AnimalsKey_get_Name (somThis->nameKey , ev) ;

    ptr3 = (string) _get_Property(anotherAnimal , ev) ;
    ptr4 = somThis->property ;

    if (strcmp(ptr1,ptr2) && strcmp(ptr3, ptr4))
        retVal = TRUE ;
    return (retVal);
}

/*
 * SOM_Scope void SOMLINK printAnimal(Animal somSelf, Environment *ev,
 *                                     Animal anAnimal)
 */

/*
 * The prototype for printAnimal was replaced by the following prototype:
 */
SOM_Scope void SOMLINK printAnimal(Animal somSelf, Environment *ev)
{
    AnimalData *somThis = AnimalGetData(somSelf);
    string ptr1;
    string ptr2;
    AnimalMethodDebug("Animal","printAnimal");

    ptr1 = (string) Animal_get_Name(somSelf , ev) ;
    ptr2 = (string) _get_Property(somSelf , ev) ;

    somPrintf ( "The %s is %s . \n", ptr1 , ptr2 ) ;

}

SOM_Scope void SOMLINK somDefaultInit(Animal somSelf, somInitCtrl* ctrl)
{
    AnimalData *somThis; /* set in BeginInitializer */
    somInitCtrl globalCtrl;
    somBooleanVector myMask;
    AnimalMethodDebug("Animal","somDefaultInit");
    Animal_BeginInitializer_somDefaultInit;

    Animal_Init_SOMObject_somDefaultInit(somSelf, ctrl);

    /*
     Just overridden with default for performance reasons
    */

}

SOM_Scope void SOMLINK somDestruct(Animal somSelf, octet doFree,
                                   somDestructCtrl* ctrl)
{
    AnimalData *somThis; /* set in BeginDestructor */
    somDestructCtrl globalCtrl;
    somBooleanVector myMask;
    AnimalMethodDebug("Animal","somDestruct");
    Animal_BeginDestructor;

    if (somThis->property != 0 )
        SOMFree (somThis->property) ;
    if (somThis->nameKey != 0 )
        _somFree(somThis->nameKey) ;

    Animal_EndDestructor;
}

```



```

----- explkey.c -----
/*
 * This file was generated by the SOM Compiler and Emitter Framework.
 * Generated using:
 *   SOM Emitter emitctm.dll: 2.41
 */

#ifndef SOM_Module_animalskey_Source
#define SOM_Module_animalskey_Source
#endif
#define AnimalsKey_Class_Source

#ifdef _MVS
#include <DD:IH(explkey)>
#else
#include "explkey.ih"
#endif

SOM_Scope void SOMLINK AnimalsKey_withName(AnimalsKey somSelf,
                                           Environment *ev,
                                           somInitCtrl* ctrl,
                                           string aName)
{
    AnimalsKeyData *somThis; /* set in BeginInitializer */
    somInitCtrl globalCtrl;
    somBooleanVector myMask;
    string ptr ;

    AnimalsKeyMethodDebug("AnimalsKey","AnimalsKey_withName");
    AnimalsKey_BeginInitializer_AnimalsKey_withName;

    AnimalsKey_Init_SOMObject_somDefaultInit(somSelf, ctrl);

    ptr = (string) SOMMalloc( strlen(aName) +1 ) ;
    strcpy (ptr, aName) ;
    somThis->name = ptr ;
}

SOM_Scope string SOMLINK get_Name(AnimalsKey somSelf, Environment *ev)
{
    AnimalsKeyData *somThis = AnimalsKeyGetData(somSelf);
    AnimalsKeyMethodDebug("AnimalsKey","get_Name");

    return (somThis->name);
}

SOM_Scope void SOMLINK somDefaultInit(AnimalsKey somSelf, somInitCtrl* ctrl)
{
    AnimalsKeyData *somThis; /* set in BeginInitializer */
    somInitCtrl globalCtrl;
    somBooleanVector myMask;
    AnimalsKeyMethodDebug("AnimalsKey","somDefaultInit");
    AnimalsKey_BeginInitializer_somDefaultInit;

    AnimalsKey_Init_SOMObject_somDefaultInit(somSelf, ctrl);

    /*
     * Just overridden with default for performance reasons
     */
}

SOM_Scope void SOMLINK somDestruct(AnimalsKey somSelf, octet doFree,
                                   somDestructCtrl* ctrl)
{
    AnimalsKeyData *somThis; /* set in BeginDestructor */

```

```

        somDestructCtrl globalCtrl;
        somBooleanVector myMask;
        AnimalsKeyMethodDebug("AnimalsKey","somDestruct");
        AnimalsKey_BeginDestructor;

        if (somThis->name != 0)
            SOMFree(somThis->name) ;

        AnimalsKey_EndDestructor;
    }

----- explops.c -----

/*
 * This file was generated by the SOM Compiler.
 * Generated using:
 *     SOM incremental update: 2.41
 */

/*
 * This file was generated by the SOM Compiler and Emitter Framework.
 * Generated using:
 *     SOM Emitter emitctm.dll: 2.41
 */

#ifndef SOM_Module_animalsops_Source
#define SOM_Module_animalsops_Source
#endif
#define AnimalsOps_Class_Source

#ifdef _MVS
#include <DD:IH(explops)>
#else
#include "explops.ih"
#endif

SOM_Scope SOMObject SOMLINK Key(AnimalsOps somSelf, Environment *ev,
                                SOMObject element)
{
    AnimalsOpsData *somThis = AnimalsOpsGetData(somSelf);
    AnimalsOpsMethodDebug("AnimalsOps","Key");
    return ((AnimalsKey )_get_Key(element, ev)) ;
}

SOM_Scope long SOMLINK KeyHash(AnimalsOps somSelf, Environment *ev,
                                SOMObject key, unsigned long value)
{
    AnimalsOpsData *somThis = AnimalsOpsGetData(somSelf);
    string ptr ;

    AnimalsOpsMethodDebug("AnimalsOps","KeyHash");
    ptr = (string) AnimalsKey_get_Name(key,ev) ;
    return ( strlen(ptr) % value ) ;
}

SOM_Scope boolean SOMLINK KeyEqual(AnimalsOps somSelf, Environment *ev,
                                    SOMObject k1, SOMObject k2)
{
    AnimalsOpsData *somThis = AnimalsOpsGetData(somSelf);
    string ptr1 ;
    string ptr2 ;
    AnimalsOpsMethodDebug("AnimalsOps","KeyEqual");
    ptr1 = (string) AnimalsKey_get_Name(k1,ev) ;
    ptr2 = (string) AnimalsKey_get_Name(k2,ev) ;

    return (0 == strcmp(ptr1,ptr2)) ;
}

```

```

SOM_Scope void SOMLINK somDefaultInit(AnimalsOps somSelf, somInitCtrl* ctrl)
{
    AnimalsOpsData *somThis; /* set in BeginInitializer */
    somInitCtrl globalCtrl;
    somBooleanVector myMask;
    AnimalsOpsMethodDebug("AnimalsOps", "somDefaultInit");
    AnimalsOps_BeginInitializer_somDefaultInit;

    AnimalsOps_Init_ISOps_somDefaultInit(somSelf, ctrl);

    /*
     Just overridden with default for performance reasons
     */
}

```

```

SOM_Scope void SOMLINK somDestruct(AnimalsOps somSelf, octet doFree,
                                   somDestructCtrl* ctrl)
{
    AnimalsOpsData *somThis; /* set in BeginDestructor */
    somDestructCtrl globalCtrl;
    somBooleanVector myMask;
    AnimalsOpsMethodDebug("AnimalsOps", "somDestruct");
    AnimalsOps_BeginDestructor;

    /*
     Just overridden with default for performance reasons
     */

    AnimalsOps_EndDestructor;
}

```

----- explini.c -----

```

#include <expllele.h>
#include <explkey.h>
#include <explops.h>

```

```

#ifdef __IBMC__
#pragma linkage(SOMInitModule, system)
#endif

```

```

SOMEXTERN void SOMLINK SOMInitModule (long majorVersion,
                                       long minorVersion,
                                       string className)
{
    AnimalNewClass(Animal_MajorVersion , Animal_MinorVersion);
    AnimalsKeyNewClass(AnimalsKey_MajorVersion , AnimalsKey_MinorVersion);
    AnimalsOpsNewClass(AnimalsOps_MajorVersion , AnimalsOps_MinorVersion);
    return;
}

```

----- explmain.c -----

```

/*-----*
| explmain.c      - Example for the use of the ISKeyBag
|                  *****
| We keep a Key Bag of our observations on animals. Elements
| handled in this Key Bag are of type animal, the key is the
| a SOMObject which maintains the name of the animal.
| This Key Bag allows us to efficiently access all observations
| on an animal.
| We use a Sequence to store the names of all extinct animals.
| At last we remove all extinct animals from the Key Bag.
|-----*
*/

```

```

#include <stdio.h>
#include <assert.h>

#include <som.h>

#include <ssglobal.h>

```

```

#include <skb.h>
#include <sseq.h>
#include <explele.h>
#include <explops.h>

#define _OK; assert(ev->_major == NO_EXCEPTION)

extern int SOM_TraceLevel;

main(int argc) {

    /*
     * Define Variables
     */

    Environment          * ev ;

    ISKeyBag              observations ;
    ISElementCursor      observationsCur1 , observationsCur2 ;

    ISSequence            extinctAnimals ;
    ISOrderedCursor        extinctAnimalsCur ;

    AnimalsOps            animalsOps ;
    Animal                anObservation ;
    Animal                anotherObservation ;
    AnimalsKey            anObservationsKey ;
    AnimalsKey            extinctAnimalsKey ;
    string                anObservationsName ;
    string                anObservationsProperty ;
    string                anotherObservationsProperty ;

    unsigned long          numberOfObservations ;
    unsigned long          numberOfAnimals ;
    unsigned long          numberOfObservationsOnAnimal ;

    boolean                more ;

    /*
     * Initialize Variables
     */

    ev                    = somGetGlobalEnvironment() ;
                        SOM_InitEnvironment(ev);

    animalsOps            = (AnimalsOps) AnimalsOpsNew() ;
    observations           = ISKeyBagNew_ISKeyBag_withOps(ev, animalsOps) ; _OK ;
    observationsCur1       = (ISElementCursor) _newElementCursor(observations, ev) ; _OK ;
    observationsCur2       = (ISElementCursor) _newElementCursor(observations, ev) ; _OK ;
    extinctAnimals         = ISSequenceNew_ISSequence_withNumber (ev ,100 ) ; _OK ;
    extinctAnimalsCur     = (ISOrderedCursor) _newOrderedCursor(extinctAnimals, ev ) ; _OK ;

    SOM_TraceLevel = 0;
    if (argc==2) SOM_TraceLevel = 1;
    somPrintf("\nAnimals Sample running...\n");

    /*
     * Collect all observations on animals made
     */

    anObservation = AnimalNew_Animal_withNameAndProperty(ev, "bear" , "heavy"); _OK ;
    _add(observations, ev, anObservation ) ; _OK ;

    anObservation = AnimalNew_Animal_withNameAndProperty(ev, "bear" , "strong"); _OK ;
    _add(observations, ev, anObservation ) ; _OK ;

```

```

anObservation = AnimalNew_Animal_withNameAndProperty(ev, "dinosaur", "heavy"); _OK ;
_add(observations, ev, anObservation) ; _OK ;

anObservation = AnimalNew_Animal_withNameAndProperty(ev, "dinosaur", "huge"); _OK ;
_add(observations, ev, anObservation) ; _OK ;

anObservation = AnimalNew_Animal_withNameAndProperty(ev, "dinosaur", "extinct"); _OK ;
_add(observations, ev, anObservation) ; _OK ;

anObservation = AnimalNew_Animal_withNameAndProperty(ev, "eagle", "black"); _OK ;
_add(observations, ev, anObservation) ; _OK ;

anObservation = AnimalNew_Animal_withNameAndProperty(ev, "eagle", "strong"); _OK ;
_add(observations, ev, anObservation) ; _OK ;

anObservation = AnimalNew_Animal_withNameAndProperty(ev, "lion", "dangerous"); _OK ;
_add(observations, ev, anObservation) ; _OK ;

anObservation = AnimalNew_Animal_withNameAndProperty(ev, "lion", "strong"); _OK ;
_add(observations, ev, anObservation) ; _OK ;


anObservation = AnimalNew_Animal_withNameAndProperty(ev, "mammoth", "longhaired"); _OK ;
_add(observations, ev, anObservation) ; _OK ;

anObservation = AnimalNew_Animal_withNameAndProperty(ev, "mammoth", "extinct"); _OK ;
_add(observations, ev, anObservation) ; _OK ;

anObservation = AnimalNew_Animal_withNameAndProperty(ev, "sabre tooth tiger", "extinct"); _OK ;
_add(observations, ev, anObservation) ; _OK ;

anObservation = AnimalNew_Animal_withNameAndProperty(ev, "zebra", "striped"); _OK ;
_add(observations, ev, anObservation) ; _OK ;

/*
 * Print content of observations
 */

somPrintf ("\nAll our observations on animals: \n" ) ;

ISElementCursor_setToFirst (observationsCurl, ev); _OK ;

for ( ; _isValid (observationsCurl, ev); )
{
    _OK ;
    anObservation = (Animal) _element( observationsCurl , ev ) ; _OK ;
    somPrintf("\n") ;
    _printAnimal(anObservation, ev ) ; _OK ;
    ISElementCursor_setToNext (observationsCurl, ev) ; _OK ;
} ;

/*
 * Print number of observations
 */

numberOfObservations = _numberOfElements(observations , ev ) ; _OK ;
somPrintf("\nNumber of observations on animals: %d \n", numberOfObservations
) ; _OK ;

/*
 * Print number of different animals
 */

numberOfAnimals = _numberOfDifferentKeys(observations , ev ) ; _OK ;
somPrintf("\nNumber of different animals observed : %d \n", numberOfAnimals
) ; _OK ;

/*
 * Create set of extinct animals

```

```

    */

ISElementCursor_setToFirst(observationsCur1,ev) ; _OK ;

do {

    anObservation = (Animal) _element( observationsCur1,ev) ; _OK ;
    anObservationsKey = (AnimalsKey) _get_Key(anObservation, ev) ; _OK ;
    anObservationsName = (string) Animal_get_Name(anObservation, ev) ; _OK ;

    numberOfObservationsOnAnimal = _numberOfElementsWithKey(observations, ev,
anObservationsKey ) ; _OK ;
    somPrintf("\nWe have %d observations on %s :\n" ,
numberOfObservationsOnAnimal, anObservationsName ) ;

    _locateElementWithKey(observations, ev, anObservationsKey,
observationsCur2); _OK ;

    do {
        _OK ;

        anotherObservation = (Animal) _element( observationsCur2,ev) ;/*
_OK ; */

        anotherObservationsProperty = (string)
_get_Property(anotherObservation, ev ) ; _OK ;

        somPrintf("      %s  \n", anotherObservationsProperty ) ;
        if (0 == strcmp(anotherObservationsProperty, "extinct"))
        {
            _add(extinctAnimals , ev , anObservationsKey ) ; _OK ;
        }

        more = _locateNextElementWithKey(observations, ev ,
anObservationsKey, observationsCur2) ;
    } while ( more ) ;

    more = _setToNextWithDifferentKey(observations, ev, observationsCur1); _OK ;

} while (more) ;

ISElementCursor_setToFirst (extinctAnimalsCur , ev) ; _OK ;

for ( ; _isValid ( extinctAnimalsCur ,ev ) ; )
{
    extinctAnimalsKey = (Animal) _element(extinctAnimalsCur, ev ) ; _OK ;
    _removeAllElementsWithKey(observations , ev, extinctAnimalsKey) ; _OK ;
    ISElementCursor_setToNext(extinctAnimalsCur,ev ) ; _OK ;

};

somPrintf( "\nAfter removing all observations on extinct animals: \n " ) ;

ISElementCursor_setToFirst (observationsCur1 , ev) ; _OK ;
for ( ; _isValid ( observationsCur1 ,ev ) ; )
{
    anotherObservation = (Animal) _element(observationsCur1, ev ) ; _OK ;
    somPrintf("\n") ;
    _printAnimal(anotherObservation , ev) ; _OK ;
    ISElementCursor_setToNext(observationsCur1 ,ev ) ; _OK ;

};

```

```

    numberOfObservations = _numberOfElements(observations, ev) ; _OK ;
    somPrintf("\nNumber of observations on animals: %d \n ",
    numberOfObservations ) ;

    numberOfAnimals = _numberOfDifferentKeys( observations , ev ) ; _OK ;
    somPrintf("\nNumber of different animals: %d \n ", numberOfAnimals ) ;

    return 0;
}

```

This program produces the following output:

```

Animals Sample running...
All our observations on animals:
The lion is strong .
The lion is dangerous .
The bear is strong .
The bear is heavy .
The zebra is striped .
The eagle is strong .
The eagle is black .
The mammoth is extinct .
The mammoth is long haired .
The dinosaur is extinct .
The dinosaur is huge .
The dinosaur is heavy .
The sabre tooth tiger is extinct .
Number of observations on animals: 13
Number of different animals observed : 7
We have 2 observations on lion :
    strong
    dangerous
We have 2 observations on bear :
    strong
    heavy
We have 1 observations on zebra :
    striped
We have 2 observations on eagle :
    strong
    black
We have 2 observations on mammoth :
    extinct
    long haired
We have 3 observations on dinosaur :
    extinct
    huge
    heavy
We have 1 observations on sabre tooth tiger :
    extinct
All observations on  extinct animals are removed.
These are the observation on not extinct animals :
The lion is strong .
The lion is dangerous .
The bear is strong .

```

The bear is heavy .
 The zebra is striped .
 The eagle is strong .
 The eagle is black .
 Number of observations on not extinct animals: 7

Number of different not extinct animals: 4

Coding Example for Set

The following program creates sets using the default class, ISSet. The odd set contains all odd numbers less than ten. The prime set contains all prime numbers less than ten. The program creates a set, oddPrime, that contains all the prime numbers less than ten that are odd, by using the intersection of odd and prime. It creates another set, evenPrime, that contains all the prime numbers less than ten that are even, by using the difference of prime and oddPrime.

When printing the sets, the program uses the applicator interface, ISApplicator. It uses the add() method to build the odd and prime sets. It uses the addIntersection() and addDifference() methods to create the oddPrime and evenPrime sets, respectively.

```
----- exp2ele.idl -----
#ifndef LONGOBJECT_IDL
#define LONGOBJECT_IDL

#include <somobj.idl>

interface LongObject :SOMObject
{

attribute   long longValue;

void        LongObject_withLong(inout somInitCtrl ctrl,
                                in long aLong) ;
void        print_longValue() ;

#ifdef __SOMIDL__
implementation
{
releaseorder : LongObject_withLong, _get_longValue,
               _set_longValue, print_longValue ;

LongObject_withLong : init ;
somDefaultInit      : override ;
somDestruct          : override ;

dllname = "exp2.dll" ;

};
#endif // __SOMIDL__

};

#endif // LONGOBJECT_IDL

----- exp2ops.idl -----
#ifndef LONGOBJECTSOPS_IDL
#define LONGOBJECTSOPS_IDL

#include <ssops.idl>
```



```

interface LongObjectsOps : ISOps
{

    #if defined __SOMIDL__
    implementation
    {
        Assign          : override ;
        Compare         : override ;
        Equal           : override ;

        somDefaultInit   : override ;
        somDestruct      : override ;

        dllname = "exp2.dll" ;

        passthru C_h_before = "#include <ssglobal.h>"
                               "#include <exp2ele.h>";
    };

    #endif //# __SOMIDL__

};

#endif //# LONGOBJECTSOPS_IDL

----- exp2appl.idl -----
#ifndef PRAPPL_IDL
#define PRAPPL_IDL

#include <sappl.idl>

interface PrintAppl : ISApplicator
{

    #if defined __SOMIDL__
    implementation
    {

        somDefaultInit   : override ;
        somDestruct      : override ;
        applyTo          : override ;

        dllname= "exp2.dll" ;

        passthru C_h_before = "#include <ssglobal.h>"
                               "#include <exp2ele.h>";

    };

    #endif // __SOMIDL__

};

#endif // PRAPPL_IDL

----- exp2ele.c -----
/*
 * This file was generated by the SOM Compiler and Emitter Framework.
 * Generated using:
 *     SOM Emitter emitctm: 2.41
 */

#ifndef SOM_Module_longobj_Source
#define SOM_Module_longobj_Source
#endif
#define LongObject_Class_Source

```

```

#ifdef _MVS
#include <DD:IH(exp2ele)>
#else
#include "exp2ele.i"
#endif

SOM_Scope void SOMLINK LongObject_withLong(LongObject somSelf,
                                           Environment *ev,
                                           somInitCtrl* ctrl,
                                           long aLong)
{
    LongObjectData *somThis; /* set in BeginInitializer */
    somInitCtrl globalCtrl;
    somBooleanVector myMask;
    LongObjectMethodDebug("LongObject","LongObject_withLong");
    LongObject_BeginInitializer_LongObject_withLong;

    LongObject_Init_SOMObject_somDefaultInit(somSelf, ctrl);

    somThis->longValue = aLong ;
}

SOM_Scope void SOMLINK print_longValue(LongObject somSelf, Environment *ev)
{
    LongObjectData *somThis = LongObjectGetData(somSelf);
    LongObjectMethodDebug("LongObject","print_longValue");

    somPrintf("%d \n", somThis->longValue ) ;
}

SOM_Scope void SOMLINK somDefaultInit(LongObject somSelf, somInitCtrl* ctrl)
{
    LongObjectData *somThis; /* set in BeginInitializer */
    somInitCtrl globalCtrl;
    somBooleanVector myMask;
    LongObjectMethodDebug("LongObject","somDefaultInit");
    LongObject_BeginInitializer_somDefaultInit;

    LongObject_Init_SOMObject_somDefaultInit(somSelf, ctrl);

    /*
     Just overridden by default for performance reasons
    */
}

SOM_Scope void SOMLINK somDestruct(LongObject somSelf, octet doFree,
                                   somDestructCtrl* ctrl)
{
    LongObjectData *somThis; /* set in BeginDestructor */
    somDestructCtrl globalCtrl;
    somBooleanVector myMask;
    LongObjectMethodDebug("LongObject","somDestruct");
    LongObject_BeginDestructor;

    /*
     Just overridden by default for performance reasons
    */

    LongObject_EndDestructor;
}

----- exp2ops.c -----

/*
 * This file was generated by the SOM Compiler.
 * Generated using:
 *     SOM incremental update: 2.41
 */

```

```

/*
 * This file was generated by the SOM Compiler and Emitter Framework.
 * Generated using:
 *   SOM Emitter emitctm.dll: 2.41
 */

#ifndef SOM_Module_longobjsops_Source
#define SOM_Module_longobjsops_Source
#endif
#define LongObjectsOps_Class_Source

#ifdef _MVS
#include <DD:IH(exp2ops)>
#else
#include "exp2ops.ih"
#endif

SOM_Scope void SOMLINK Assign(LongObjectsOps somSelf, Environment *ev,
                              SOMObject e1, SOMObject e2)
{
    /* LongObjectsOpsData *somThis = LongObjectsOpsGetData(somSelf); */

    long val2 ;

    LongObjectsOpsMethodDebug("LongObjectsOps","Assign");
    val2 = __get_longValue( e2, ev ) ;
    __set_longValue( e1, ev , val2 ) ;
}

SOM_Scope long SOMLINK Compare(LongObjectsOps somSelf, Environment *ev,
                               SOMObject e1, SOMObject e2)
{
    /* LongObjectsOpsData *somThis = LongObjectsOpsGetData(somSelf); */

    long val1 ;
    long val2 ;

    LongObjectsOpsMethodDebug("LongObjectsOps","Compare");

    val1 = __get_longValue( e1, ev ) ;
    val2 = __get_longValue( e2, ev ) ;
    return (val1 - val2) ;
}

SOM_Scope boolean SOMLINK Equal(LongObjectsOps somSelf, Environment *ev,
                                SOMObject e1, SOMObject e2)
{
    /* LongObjectsOpsData *somThis = LongObjectsOpsGetData(somSelf); */

    long val1 ;
    long val2 ;

    LongObjectsOpsMethodDebug("LongObjectsOps","Equal");

    val1 = __get_longValue( e1, ev ) ;
    val2 = __get_longValue( e2, ev ) ;
    return (val1 == val2) ;
}

SOM_Scope void SOMLINK somDefaultInit(LongObjectsOps somSelf,
                                       somInitCtrl* ctrl)
{
    LongObjectsOpsData *somThis; /* set in BeginInitializer */
    somInitCtrl globalCtrl;
    somBooleanVector myMask;
    LongObjectsOpsMethodDebug("LongObjectsOps","somDefaultInit");
    LongObjectsOps_BeginInitializer_somDefaultInit;

    LongObjectsOps_Init_ISOps_somDefaultInit(somSelf, ctrl);
}

```

```

        /*
        Just overridden with default for performance reasons
        */
    }

SOM_Scope void SOMLINK somDestruct(LongObjectsOps somSelf, octet doFree,
                                   somDestructCtrl* ctrl)
{
    LongObjectsOpsData *somThis; /* set in BeginDestructor */
    somDestructCtrl globalCtrl;
    somBooleanVector myMask;
    LongObjectsOpsMethodDebug("LongObjectsOps", "somDestruct");
    LongObjectsOps_BeginDestructor;

    /*
    Just overridden with default for performance reasons
    */

    LongObjectsOps_EndDestructor;
}

----- exp2appl.c -----
/*
 * This file was generated by the SOM Compiler and Emitter Framework.
 * Generated using:
 *     SOM Emitter emitctm.dll: 2.41
 */

#ifndef SOM_Module_prappl_Source
#define SOM_Module_prappl_Source
#endif
#define PrintAppl_Class_Source

#ifdef _MVS
#include <DD:IH(exp2appl)>
#else
#include "exp2appl.i.h"
#endif

SOM_Scope void SOMLINK somDefaultInit(PrintAppl somSelf, somInitCtrl* ctrl)
{
    PrintApplData *somThis; /* set in BeginInitializer */
    somInitCtrl globalCtrl;
    somBooleanVector myMask;
    PrintApplMethodDebug("PrintAppl", "somDefaultInit");
    PrintAppl_BeginInitializer_somDefaultInit;

    PrintAppl_Init_ISApplicator_somDefaultInit(somSelf, ctrl);

    /*
    Just overridden for performance reasons
    */
}

SOM_Scope void SOMLINK somDestruct(PrintAppl somSelf, octet doFree,
                                   somDestructCtrl* ctrl)
{
    PrintApplData *somThis; /* set in BeginDestructor */
    somDestructCtrl globalCtrl;
    somBooleanVector myMask;
    PrintApplMethodDebug("PrintAppl", "somDestruct");
    PrintAppl_BeginDestructor;

    /*
    Just overridden for performance reasons
    */
}

```

```

        PrintAppl_EndDestructor;
    }

SOM_Scope boolean SOMLINK applyTo(PrintAppl somSelf, Environment *ev,
                                   SOMObject element)
{
    /* PrintApplData *somThis = PrintApplGetData(somSelf); */
    PrintApplMethodDebug("PrintAppl","applyTo");

    LongObject_print_longValue(element , ev ) ;

    return (TRUE) ;

}

----- exp2ini.c -----
#include <exp2ele.h>
#include <exp2ops.h>
#include <exp2appl.h>

#ifdef __IBMC__
#pragma linkage(SOMInitModule, system)
#endif

SOMEXTERN void SOMLINK SOMInitModule (long majorVersion,
                                       long minorVersion,
                                       string className)
{
    LongObjectNewClass(LongObject_MajorVersion , LongObject_MinorVersion);
    LongObjectsOpsNewClass(LongObjectsOps_MajorVersion , LongObjectsOps_MinorVersion);
    PrintApplNewClass(PrintAppl_MajorVersion , PrintAppl_MinorVersion);
    return;
}

----- exp2main.c -----
#include <stdio.h>
#include <assert.h>

#include <som.h>

#include <sSGlobal.h>

#include <sset.h>
#include <exp2ele.h>
#include <exp2ops.h>
#include <exp2appl.h>

#define _OK ; assert(ev->_major == NO_EXCEPTION)

#define _ERR ; assert(ev->_major == USER_EXCEPTION); \
    ev->_major = NO_EXCEPTION; \
    somPrintf("\nexception: %s\n",somExceptionId(ev));

extern int SOM_TraceLevel ;

typedef ISset LongSet ;

void List (char * , Environment * , LongSet ) ;

int main (int argc)
{

    Environment * ev ;

```

```

LongSet      odd, prime ;
LongSet      oddPrime , evenPrime;
LongObjectsOps longObjectsOpsOdd ;
LongObjectsOps longObjectsOpsPrime ;
LongObjectsOps longObjectsOpsOddPrime ;
LongObjectsOps longObjectsOpsEvenPrime ;

LongObject   One ;
LongObject   Two ;
LongObject   Three ;
LongObject   Five ;
LongObject   Seven ;
LongObject   Nine ;

long          number ;

somPrintf("\nEvenOdd Sample running ... \n");

ev            = (Environment *) somGetGlobalEnvironment() ;
              SOM_InitEnvironment(ev) ;

longObjectsOpsOdd
    = (LongObjectsOps) LongObjectsOpsNew() ;
longObjectsOpsPrime
    = (LongObjectsOps) LongObjectsOpsNew() ;
longObjectsOpsOddPrime
    = (LongObjectsOps) LongObjectsOpsNew() ;
longObjectsOpsEvenPrime
    = (LongObjectsOps) LongObjectsOpsNew() ;

odd           = (LongSet)   ISSetNew_ISSet_withOps(ev, longObjectsOpsOdd) ; _OK ;
prime         = (LongSet)   ISSetNew_ISSet_withOps(ev, longObjectsOpsPrime) ; _OK ;

oddPrime      = (LongSet)   ISSetNew_ISSet_withOps(ev, longObjectsOpsOddPrime) ; _OK ;
evenPrime     = (LongSet)   ISSetNew_ISSet_withOps(ev, longObjectsOpsEvenPrime) ; _OK ;

One           = (LongObject) LongObjectNew_LongObject_withLong(ev, 1) ; _OK ;
Two           = (LongObject) LongObjectNew_LongObject_withLong(ev, 2) ; _OK ;
Three         = (LongObject) LongObjectNew_LongObject_withLong(ev, 3) ; _OK ;
Five          = (LongObject) LongObjectNew_LongObject_withLong(ev, 5) ; _OK ;
Seven         = (LongObject) LongObjectNew_LongObject_withLong(ev, 7) ; _OK ;
Nine          = (LongObject) LongObjectNew_LongObject_withLong(ev, 9) ; _OK ;

SOM_TraceLevel = 0 ;
if(argc==2) SOM_TraceLevel = 1 ;

_add(odd, ev, One); _OK ;
number = _numberOfElements(odd, ev); _OK ;
_add(odd, ev, Three); _OK ;
number = _numberOfElements(odd, ev); _OK ;
_add(odd, ev, Five); _OK ;
number = _numberOfElements(odd, ev); _OK ;
_add(odd, ev, Seven); _OK ;
number = _numberOfElements(odd, ev); _OK ;
_add(odd, ev, Nine); _OK ;
number = _numberOfElements(odd, ev); _OK ;

List("Odds less than 10: ", ev, odd) ; _OK ;

_add(prime, ev, Two); _OK ;
number = _numberOfElements(odd, ev); _OK ;
_add(prime, ev, Three); _OK ;
number = _numberOfElements(odd, ev); _OK ;
_add(prime, ev, Five); _OK ;
number = _numberOfElements(odd, ev); _OK ;
_add(prime, ev, Seven); _OK ;
number = _numberOfElements(odd, ev); _OK ;

List("Primes less than 10: ", ev, prime) ; _OK ;

```

```

_addIntersection(oddPrime, ev, odd , prime) ; _OK ;

List("Odd Primes less than 10:  ", ev ,oddPrime);

_addDifference(evenPrime, ev, prime, oddPrime) ; _OK ;

List("Even primes less than 10:  ", ev , evenPrime) ; _OK ;

return (0) ;

}

void List (char * aMessage, Environment * ev, LongSet aLongSet )
{
PrintAppl      printAppl ;
printAppl      = PrintApplNew() ;

somPrintf("%s \n" , aMessage) ;
_allElementsDo(aLongSet,ev,printAppl); _OK;
somPrintf("\n") ;

}

```

This program produces the following output:

```

EvenOdd Sample running ...
Odds less than 10:
1
3
5
7
9
Primes less than 10:
2
3
5
7
Odd Primes less than 10:
3
5
7
Even primes less than 10:
2

```

Coding Example for Sorted Set

The following program uses the default class, `ISSortedSet`, to create sorted lists of planets with different properties. The program stores all planets in our solar system, all heavy planets in our solar system, all bright planets in our solar system, and all heavy or bright planets in our solar system in a number of sorted sets. Each set sorts the planets by its distance from the sun.

It uses the `allElementsDo()` function to display the planets in each collection and the `unionWith()` function when creating the bright-or-heavy planets category.

```
----- exp5ele.idl -----
#ifndef PLANET_IDL
#define PLANET_IDL

#include <somobj.idl>    ///< Get general parent class definition

interface Planet : SOMObject {

    void Planet_withNameDistanceMassAndBrightness
        (inout somInitCtrl ctrl, in string aName, in float aDist, in float aMass, in float aBright);
    // Constructor to define a planet from name, distance, mass and brightness

    boolean equalPlanets (in Planet anotherPlanet);
    // For a Set we need to provide element equality.

    long isSmaller (in Planet anotherPlanet);
    // For a Sorted Set we need to provide element comparison.

    string name();
    // Get method for the name of the planet

    boolean isHeavy();
    // Method to determine, if the planet is heavier than earth

    boolean isBright();
    // Method to determine, if the planet is bright

#ifdef __PRIVATE__
    string get_plname ();
    float get_dist ();
    // Private methods to get instance variables
#endif

#if defined __SOMIDL__
implementation
{

#ifdef __PRIVATE__
    // Define instance variables
    string pname; // name of the planet
    float dist;   // distance of the planet from the sun
    float mass;   // mass of the planet (multiples of the earths mass)
    float bright; // brightness of the planet
#endif

    releaseorder : Planet_withNameDistanceMassAndBrightness,
                  equalPlanets,
                  isSmaller,
                  name,
                  isHeavy,
                  isBright,
                  get_plname,
                  get_dist;

    Planet_withNameDistanceMassAndBrightness : init;

    somDefaultInit      : override ;
    somDestruct         : override ;

    dllname = "planet.dll" ;

};
#endif    ///< SOM_IDL

};
```



```

#endif    ///< PLANET_IDL
----- exp5ops.idl -----
#ifndef PLANETOPS_IDL
#define PLANETOPS_IDL

#include <ssops.idl>

interface PlanetOps : ISOps
{

#if defined __SOMIDL__
implementation
{
    Compare          : override ;
    Equal            : override ;

    somDefaultInit    : override ;
    somDestruct        : override ;

    dllname = "exp5.dll" ;

    passthru C_h_before = "#include <ssglobal.h>"
                          "#include <exp5ele.h>" ;
};

#endif    ///< __SOMIDL__

};
#endif    ///< PLANETOPS_IDL

----- exp5appl.idl -----
#ifndef SAYPLANETNAME_IDL
#define SAYPLANETNAME_IDL

#include <sappl.idl>    ///< Get general parent class definition
#include <exp5ele.idl>    ///< Get planet class definition

interface SayPlanetName : ISApplicator {

#if defined __SOMIDL__
implementation
{
    somDefaultInit    : override ;
    somDestruct        : override ;
    applyTo           : override ;

    dllname = "exp5.dll" ;

    passthru C_h_before = "#include <exp5ele.h>";

};

#endif    ///< SOM_IDL

};

#endif    ///< SAYPLANETNAME_IDL

----- exp5ele.c -----

/*
 * This file was generated by the SOM Compiler.
 * Generated using:
 *     SOM incremental update: 2.41
 */

/*
 * This file was generated by the SOM Compiler and Emitter Framework.
 * Generated using:
 *     SOM Emitter emitctm.dll: 2.41

```

```

*/

#ifndef SOM_Module_planet_Source
#define SOM_Module_planet_Source
#endif
#define Planet_Class_Source

#ifdef _MVS
#include <DD:IH(exp5ele)>
#else
#include "exp5ele.i"
#endif

/*
 * Constructor to define a planet from name, distance, mass and brightness
 */
SOM_Scope void SOMLINK Planet_withNameDistanceMassAndBrightness(Planet somSelf,
                                                                Environment *ev,
                                                                somInitCtrl* ctrl,
                                                                string aName,
                                                                float aDist,
                                                                float aMass,
                                                                float aBright)

{
    string ptr;
    PlanetData *somThis; /* set in BeginInitializer */
    somInitCtrl globalCtrl;
    somBooleanVector myMask;
    PlanetMethodDebug("Planet", "Planet_withNameDistanceMassAndBrightness");
    Planet_BeginInitializer_Planet_withNameDistanceMassAndBrightness;

    Planet_Init_SOMObject_somDefaultInit(somSelf, ctrl);

    /*
     * local Planet initialization code added by programmer
     */

    somThis -> dist    = aDist;
    somThis -> mass    = aMass;
    somThis -> bright  = aBright;

    ptr = (string) SOMMalloc(strlen(aName) + 1 ) ;
    strcpy (ptr, aName) ;
    somThis->pName = ptr ;
}

/*
 * For a Set we need to provide element equality.
 */
SOM_Scope boolean SOMLINK equalPlanets(Planet somSelf, Environment *ev,
                                         Planet anotherPlanet)
{
    boolean returnValue = FALSE ;
    string  firstName ;
    string  secondName ;

    PlanetData *somThis = PlanetGetData(somSelf);
    PlanetMethodDebug("Planet", "equalPlanets");

    firstName = get_pName (somSelf , ev) ;
    secondName = get_pName (anotherPlanet , ev) ;

    /* Return statement to be customized: */
    if (strcmp(firstName, secondName) )
        returnValue = TRUE ;

    return (returnValue);
}

```

```

/*
 * For a Sorted Set we need to provide element comparision.
 */

SOM_Scope long  SOMLINK isSmaller(Planet somSelf, Environment *ev,
                                Planet anotherPlanet)
{
    float distance;
    PlanetData *somThis = PlanetGetData(somSelf);
    PlanetMethodDebug("Planet","isSmaller");

    /* Compare the distances of the planets */
    distance = get_dist (anotherPlanet, ev);
    if ( (somThis -> dist) < distance) { return (-1); }
    else { if ( (somThis -> dist) > distance) { return (1); }
          else { return (0); }
    } /* endif */
}

/*
 * Get method for the name of the planet
 */

SOM_Scope string  SOMLINK name(Planet somSelf, Environment *ev)
{
    PlanetData *somThis = PlanetGetData(somSelf);
    PlanetMethodDebug("Planet","name");

    /* Return the name of the planet */
    return ( get_plname (somSelf, ev) );
}

/*
 * Method to determine, if the planet is heavier than earth
 */

SOM_Scope boolean  SOMLINK isHeavy(Planet somSelf, Environment *ev)
{
    PlanetData *somThis = PlanetGetData(somSelf);
    PlanetMethodDebug("Planet","isHeavy");

    /* Check if the mass of the planet is greater than 1.0 */
    return ( (somThis -> mass) > 1.0 );
}

/*
 * Method to determine, if the planet is bright
 */

SOM_Scope boolean  SOMLINK isBright(Planet somSelf, Environment *ev)
{
    PlanetData *somThis = PlanetGetData(somSelf);
    PlanetMethodDebug("Planet","isBright");

    /* Check the brightness of the planet */
    return ( (somThis -> bright) < 0.0 );
}

SOM_Scope string  SOMLINK get_plname(Planet somSelf, Environment *ev)
{
    PlanetData *somThis = PlanetGetData(somSelf);
    PlanetMethodDebug("Planet","get_plname");

    /* Return the name of the planet */
    return (somThis -> plname);
}

/*
 * The prototype for get_dist was replaced by the following prototype:

```

```

    */
/*
 * Private methods to get instance variables
 */

SOM_Scope float  SOMLINK get_dist(Planet somSelf, Environment *ev)
{
    PlanetData *somThis = PlanetGetData(somSelf);
    PlanetMethodDebug("Planet","get_dist");

    /* Return the distance of the planet */
    return (somThis -> dist);
}

SOM_Scope void SOMLINK somDefaultInit(Planet somSelf, somInitCtrl* ctrl)
{
    PlanetData *somThis; /* set in BeginInitializer */
    somInitCtrl globalCtrl;
    somBooleanVector myMask;
    PlanetMethodDebug("Planet","somDefaultInit");
    Planet_BeginInitializer_somDefaultInit;

    Planet_Init_SOMObject_somDefaultInit(somSelf, ctrl);

    /*
     * local Planet initialization code added by programmer
     */
}

SOM_Scope void SOMLINK somDestruct(Planet somSelf, octet doFree,
                                   somDestructCtrl* ctrl)
{
    PlanetData *somThis; /* set in BeginDestructor */
    somDestructCtrl globalCtrl;
    somBooleanVector myMask;
    PlanetMethodDebug("Planet","somDestruct");
    Planet_BeginDestructor;

    /* Free the pname */
    if (somThis->pname != 0 )
        SOMFree (somThis->pname) ;

    Planet_EndDestructor;
}

----- exp5ops.c -----
/*
 * This file was generated by the SOM Compiler and Emitter Framework.
 * Generated using:
 *   SOM Emitter emitctm.dll: 2.41
 */

#ifdef SOM_Module_planetops_Source
#define SOM_Module_planetops_Source
#endif
#define PlanetOps_Class_Source

#ifdef _MVS
#include <DD:IH(exp5ops)>
#else
#include "exp5ops.ih"
#endif

SOM_Scope long  SOMLINK Compare(PlanetOps somSelf, Environment *ev,
                               SOMObject e1, SOMObject e2)
{
    /* PlanetOpsData *somThis = PlanetOpsGetData(somSelf); */

```

```

        PlanetOpsMethodDebug("PlanetOps","Compare");

        /* return (PlanetOps_parent_ISOps_Compare(somSelf, ev, e1, e2)); */

        return ( Planet_isSmaller ( e1, ev, e2));
    }

SOM_Scope boolean SOMLINK Equal(PlanetOps somSelf, Environment *ev,
                                SOMObject e1, SOMObject e2)
{
    /* PlanetOpsData *somThis = PlanetOpsGetData(somSelf); */
    PlanetOpsMethodDebug("PlanetOps","Equal");

    /* return (PlanetOps_parent_ISOps_Equal(somSelf, ev, e1, e2)); */

    return ( Planet_equalPlanets ( e1, ev, e2));
}

SOM_Scope void SOMLINK somDefaultInit(PlanetOps somSelf, somInitCtrl* ctrl)
{
    PlanetOpsData *somThis; /* set in BeginInitializer */
    somInitCtrl globalCtrl;
    somBooleanVector myMask;
    PlanetOpsMethodDebug("PlanetOps","somDefaultInit");
    PlanetOps_BeginInitializer_somDefaultInit;

    PlanetOps_Init_ISOps_somDefaultInit(somSelf, ctrl);

    /*
     * local PlanetOps initialization code added by programmer
     */
}

SOM_Scope void SOMLINK somDestruct(PlanetOps somSelf, octet doFree,
                                    somDestructCtrl* ctrl)
{
    PlanetOpsData *somThis; /* set in BeginDestructor */
    somDestructCtrl globalCtrl;
    somBooleanVector myMask;
    PlanetOpsMethodDebug("PlanetOps","somDestruct");
    PlanetOps_BeginDestructor;

    /*
     * local PlanetOps deinitialization code added by programmer
     */

    PlanetOps_EndDestructor;
}

----- exp5app1.c -----

/*
 * This file was generated by the SOM Compiler.
 * Generated using:
 *     SOM incremental update: 2.41
 */

/*
 * This file was generated by the SOM Compiler and Emitter Framework.
 * Generated using:
 *     SOM Emitter emitctm.dll: 2.41
 */

#ifdef SOM_Module_sayplanetname_Source
#define SOM_Module_sayplanetname_Source
#endif
#define SayPlanetName_Class_Source

```

```

#ifdef _MVS
#include <DD:IH(exp5appl)>
#else
#include "exp5appl.i.h"
#endif

SOM_Scope void SOMLINK somDefaultInit(SayPlanetName somSelf,
                                       somInitCtrl* ctrl)
{
    SayPlanetNameData *somThis; /* set in BeginInitializer */
    somInitCtrl globalCtrl;
    somBooleanVector myMask;
    SayPlanetNameMethodDebug("SayPlanetName","somDefaultInit");
    SayPlanetName_BeginInitializer_somDefaultInit;

    SayPlanetName_Init_ISApplicator_somDefaultInit(somSelf, ctrl);

    /*
     * local SayPlanetName initialization code added by programmer
     */
}

SOM_Scope void SOMLINK somDestruct(SayPlanetName somSelf, octet doFree,
                                   somDestructCtrl* ctrl)
{
    SayPlanetNameData *somThis; /* set in BeginDestructor */
    somDestructCtrl globalCtrl;
    somBooleanVector myMask;
    SayPlanetNameMethodDebug("SayPlanetName","somDestruct");
    SayPlanetName_BeginDestructor;

    /*
     * local SayPlanetName deinitialization code added by programmer
     */

    SayPlanetName_EndDestructor;
}

SOM_Scope boolean SOMLINK applyTo(SayPlanetName somSelf, Environment *ev,
                                   SOMObject element)
{
    /* SayPlanetNameData *somThis = SayPlanetNameGetData(somSelf); */
    string planetsName ;
    SayPlanetNameMethodDebug("SayPlanetName","applyTo");

    /* return (SayPlanetName_parent_ISApplicator_applyTo(somSelf,
                                                         ev, element));
    */
    planetsName = Planet_name (element, ev) ;
    somPrintf(" Planet: %s \n ", planetsName );
    return (TRUE);
}

----- exp5ini.c -----

#include <exp5ele.h>
#include <exp5ops.h>
#include <exp5appl.h>

#ifdef __IBMC__
#pragma linkage(SOMInitModule, system)
#endif

SOMEXTERN void SOMLINK SOMInitModule (long majorVersion,
                                       long minorVersion,
                                       string className)
{
    PlanetNewClass(Planet_MajorVersion,
                  Planet_MinorVersion);
    PlanetOpsNewClass(PlanetOps_MajorVersion,
                     PlanetOps_MinorVersion);
}

```

```

        SayPlanetNameNewClass(SayPlanetName_MajorVersion,
                               SayPlanetName_MinorVersion);
    return;
}

----- exp5main.c -----
/*-----*
| exp5main.c - All known planets are handled in a Sorted Set.
|               |
|               | This example creates several sorted sets of planets.
|               | The sort order is based on each planets distance from
|               | the sun.
|-----*
|-----*

#include <stdio.h>
#include <assert.h>

#include <som.h>

#include <ssglobal.h>

#include <sss.h>
#include <exp5ele.h>
#include <exp5ops.h>
#include <exp5appl.h>

#define _OK; assert(ev -> _major == NO_EXCEPTION)

extern int SOM_TraceLevel;

int main(int argc)    {

    /*
     * Define Variables
     */

    Environment        * ev ;
    ISSortedSet         allPlanets, heavyPlanets, brightPlanets ;
    ISOrderedCursor     aPlanetCursor ;
    SayPlanetName       showPlanet;
    Planet              aPlanet;
    PlanetOps            planetOps , heavyPlanetOps , brightPlanetOps ;

    int                 number;

    /*
     * Initialize Variables
     */

    SOM_TraceLevel = 0;
    if (argc == 2) SOM_TraceLevel = 1;

    ev                = somGetGlobalEnvironment() ;
                     SOM_InitEnvironment(ev);

    planetOps          = (PlanetOps) PlanetOpsNew() ;
    heavyPlanetOps     = (PlanetOps) PlanetOpsNew() ;
    brightPlanetOps    = (PlanetOps) PlanetOpsNew() ;

    showPlanet         = (SayPlanetName) SayPlanetNameNew() ;

    allPlanets         = ISSortedSetNew_ISSortedSet_withOps(ev, planetOps) ; _OK ;
    heavyPlanets       = ISSortedSetNew_ISSortedSet_withOps(ev, heavyPlanetOps) ; _OK ;
    brightPlanets      = ISSortedSetNew_ISSortedSet_withOps(ev, brightPlanetOps) ; _OK ;

    aPlanetCursor      = (ISOrderedCursor) _newElementCursor(allPlanets, ev) ; _OK ;

    /*
     * Define the Planets and fill the allPlanets set
     */

    aPlanet = PlanetNew_Planet_withNameDistanceMassAndBrightness

```

```

        (ev, "Earth", 149.60, 1.0000, 99.9); _OK ;
    _add (allPlanets, ev, aPlanet); _OK ;

    /* "Dummy Earth" to check if elements with an equal distance are not added. */
    aPlanet = PlanetNew_Planet_withNameDistanceMassAndBrightness
        (ev, "Earth2", 149.60, 1.0000, 99.9); _OK ;
    _add (allPlanets, ev, aPlanet); _OK ;

    aPlanet = PlanetNew_Planet_withNameDistanceMassAndBrightness
        (ev, "Jupiter", 778.3, 317.818, -2.4); _OK ;
    _add (allPlanets, ev, aPlanet); _OK ;

    aPlanet = PlanetNew_Planet_withNameDistanceMassAndBrightness
        (ev, "Mars", 227.9, 0.1078, -1.9); _OK ;
    _add (allPlanets, ev, aPlanet); _OK ;

    aPlanet = PlanetNew_Planet_withNameDistanceMassAndBrightness
        (ev, "Mercury", 57.91, 0.0558, -0.2); _OK ;
    _add (allPlanets, ev, aPlanet); _OK ;

    aPlanet = PlanetNew_Planet_withNameDistanceMassAndBrightness
        (ev, "Neptun", 4498.0, 17.216, +7.6); _OK ;
    _add (allPlanets, ev, aPlanet); _OK ;

    aPlanet = PlanetNew_Planet_withNameDistanceMassAndBrightness
        (ev, "Pluto", 5910.0, 0.18, +14.7); _OK ;
    _add (allPlanets, ev, aPlanet); _OK ;

    aPlanet = PlanetNew_Planet_withNameDistanceMassAndBrightness
        (ev, "Saturn", 1428.0, 95.112, +0.8); _OK ;
    _add (allPlanets, ev, aPlanet); _OK ;

    aPlanet = PlanetNew_Planet_withNameDistanceMassAndBrightness
        (ev, "Uranus", 2872.0, 14.517, +5.8); _OK ;
    _add (allPlanets, ev, aPlanet); _OK ;

    aPlanet = PlanetNew_Planet_withNameDistanceMassAndBrightness
        (ev, "Venus", 108.21, 0.8148, -4.1); _OK ;
    _add (allPlanets, ev, aPlanet); _OK ;

    /*
    * Check for heavy and bright planets
    */

    ISOrderedCursor_setToFirst (aPlanetCursor, ev); _OK ;

    for ( ; _isValid (aPlanetCursor, ev); )
    {
        _OK ;
        if ( _isHeavy( (Planet) _elementAt(allPlanets, ev, aPlanetCursor), ev) )
            _add(heavyPlanets, ev, ( (Planet) _elementAt(allPlanets, ev, aPlanetCursor)) ); _OK ;

        if ( _isBright( (Planet) _elementAt(allPlanets, ev, aPlanetCursor), ev) )
            _add(brightPlanets, ev, ( (Planet) _elementAt(allPlanets, ev, aPlanetCursor)) ); _OK ;

        ISOrderedCursor_setToNext (aPlanetCursor, ev) ; _OK ;
    }

    /*
    * Print the results
    */

    somPrintf("\nAll Planets: \n"); _OK ;
    number = _numberOfElements (allPlanets, ev); _OK ;
    somPrintf("\nNumber of Planets: %d \n", number); _OK ;

    _allElementsDo(allPlanets, ev, showPlanet); _OK ;

    somPrintf("\nHeavy Planets: \n"); _OK ;
    number = _numberOfElements (heavyPlanets, ev); _OK ;
    somPrintf("\nNumber of Planets: %d \n", number); _OK ;

```



```

    _allElementsDo(heavyPlanets, ev, showPlanet); _OK ;

    somPrintf("\nBright Planets: \n"); _OK ;
    number = _numberOfElements (brightPlanets, ev); _OK ;
    somPrintf("\nNumber of Planets: %d \n", number); _OK ;

    _allElementsDo(brightPlanets, ev, showPlanet); _OK ;

    somPrintf("\nBright or Heavy Planets: \n"); _OK ;
    _unionWith(brightPlanets, ev, heavyPlanets); _OK ;
    number = _numberOfElements (brightPlanets, ev); _OK ;
    somPrintf("\nNumber of Planets: %d \n", number); _OK ;

    _allElementsDo(brightPlanets, ev, showPlanet); _OK ;

    somPrintf("\n \n \n"); _OK ;
    somPrintf("\nDid you notice that all these Sets are sorted"); _OK ;
    somPrintf("\nin the same order"); _OK ;
    somPrintf("\n (distance of planet from sun) ? \n \n \n"); _OK ;

    return 0; _OK ;
}

```

This program produces the following output:

All Planets:

Number of Planets: 9

Planet: Mercury
 Planet: Venus
 Planet: Earth
 Planet: Mars
 Planet: Jupiter
 Planet: Saturn
 Planet: Uranus
 Planet: Neptun
 Planet: Pluto

Heavy Planets:

Number of Planets: 4

Planet: Jupiter
 Planet: Saturn
 Planet: Uranus
 Planet: Neptun

Bright Planets:

Number of Planets: 4

Planet: Mercury
 Planet: Venus
 Planet: Mars
 Planet: Jupiter

Bright or Heavy Planets:

Number of Planets: 7

Planet: Mercury
 Planet: Venus
 Planet: Mars
 Planet: Jupiter
 Planet: Saturn

Planet: Uranus
Planet: Neptun

Did you notice that all these Sets are sorted
in the same order
(distance of planet from sun) ?

Coding Example for Stack

The following program creates two stacks (Stack1 and Stack2) using the default class, ISStack. It adds a number of words to Stack1, removes them from Stack1, adds them to Stack2, and finally removes them from Stack2 so that they can be printed. The push() and pop() functions are used for adding and removing elements, respectively.

Between these stack operations the stacks are printed. To prevent the stack from changing during printing, the program uses the constant version of the iterator class, IConstantIterator with the allElementsDo() function. The words print in the same order as they were originally added to Stack1.

Because of the nature of the stack class, the program must use the constant iterator class, IConstantIterator, when printing the stacks. It uses the push() and pop() functions for adding and removing elements, respectively. The allElementsDo() function is used when the collection is printed.

```
----- exp4ele.idl -----

#ifndef LONGOBJECT_IDL
#define LONGOBJECT_IDL

#include <somobj.idl>

interface StringObject :SOMObject
{

attribute   string stringValue;

void        StringObject_withString(inout somInitCtrl ctrl,
                                   in string aString) ;
void        print_stringValue() ;

#ifdef __SOMIDL__
implementation
{
releaseorder : StringObject_withString,
               _get_stringValue,
               _set_stringValue,
               print_stringValue ;

StringObject_withString : init ;
stringValue             : noget , noset ;
somDefaultInit           : override ;
somDestruct              : override ;

dllname = "exp4.dll" ;

passthru C_h_before = "#include <string.h>" ;

```

```

};
#endif // __SOMIDL__

};

#endif // LONGOBJECT_IDL

----- exp4appl.idl -----
#ifndef PRAPPL_IDL
#define PRAPPL_IDL

#include <sappl.idl>

interface PrintAppl : ISApplicator
{

#if defined __SOMIDL__
implementation
{

    somDefaultInit      : override ;
    somDestruct         : override ;
    applyTo             : override ;

    dllname= "exp4.dll" ;

    passthru C_h_before = "#include <ssglobal.h>"
                        "#include <exp4ele.h>";

};
#endif // __SOMIDL__

};

#endif // PRAPPL_IDL

----- exp4ele.c -----

/*
 * This file was generated by the SOM Compiler and Emitter Framework.
 * Generated using:
 *     SOM Emitter emitctm.dll: 2.41
 */

#ifndef SOM_Module_stringobj_Source
#define SOM_Module_stringobj_Source
#endif
#define StringObject_Class_Source

#ifdef _MVS
#include <DD:IH(exp4ele)>
#else
#include "exp4ele.ih"
#endif

/*
 *Method from the IDL attribute statement:
 *"attribute string stringValue"
 */

SOM_Scope string  SOMLINK _get_stringValue(StringObject somSelf,
                                           Environment *ev)
{
    StringObjectData *somThis = StringObjectGetData(somSelf);
    StringObjectMethodDebug("StringObject", "_get_stringValue");

    /* Return statement to be customized: */

```

```

        return (somThis->stringValue) ;
    }

    /*
    *Method from the IDL attribute statement:
    *"attribute string stringValue"
    */

    SOM_Scope void SOMLINK _set_stringValue(StringObject somSelf,
                                           Environment *ev,
                                           string stringValue)
    {
        StringObjectData *somThis = StringObjectGetData(somSelf);
        StringObjectMethodDebug("StringObject", "_set_stringValue");

        if(somThis->stringValue != 0)
            SOMFree(somThis->stringValue) ;

        somThis->stringValue = (string) SOMMalloc(strlen (stringValue) +1 );
        strcpy ( somThis->stringValue , stringValue) ;
    }

    SOM_Scope void SOMLINK StringObject_withString(StringObject somSelf,
                                                    Environment *ev,
                                                    somInitCtrl* ctrl,
                                                    string aString)
    {
        StringObjectData *somThis; /* set in BeginInitializer */
        somInitCtrl globalCtrl;
        somBooleanVector myMask;
        StringObjectMethodDebug("StringObject", "StringObject_withString");
        StringObject_BeginInitializer_StringObject_withString;
        StringObject_Init_SOMObject_somDefaultInit(somSelf, ctrl);

        somThis->stringValue = (string) SOMMalloc(strlen (aString) +1 );
        strcpy ( somThis->stringValue , aString) ;
    }

    SOM_Scope void SOMLINK print_stringValue(StringObject somSelf,
                                              Environment *ev)
    {
        StringObjectData *somThis = StringObjectGetData(somSelf);
        StringObjectMethodDebug("StringObject", "print_stringValue");
        somPrintf("%s \n" , somThis->stringValue) ;
    }

    SOM_Scope void SOMLINK somDefaultInit(StringObject somSelf,
                                           somInitCtrl* ctrl)
    {
        StringObjectData *somThis; /* set in BeginInitializer */
        somInitCtrl globalCtrl;
        somBooleanVector myMask;
        StringObjectMethodDebug("StringObject", "somDefaultInit");
        StringObject_BeginInitializer_somDefaultInit;

        StringObject_Init_SOMObject_somDefaultInit(somSelf, ctrl);

        /*
        Just overridden with Default for performance reasons
        */
    }

    SOM_Scope void SOMLINK somDestruct(StringObject somSelf, octet doFree,
                                       somDestructCtrl* ctrl)
    {
        StringObjectData *somThis; /* set in BeginDestructor */
        somDestructCtrl globalCtrl;
    }

```

```

        somBooleanVector myMask;
        StringObjectMethodDebug("StringObject","somDestruct");
        StringObject_BeginDestructor;

        if(somThis->stringValue !=0)
            SOMFree(somThis->stringValue) ;

        StringObject_EndDestructor;
    }

----- exp4appl.c -----
/*
 * This file was generated by the SOM Compiler and Emitter Framework.
 * Generated using:
 *     SOM Emitter emitctm.dll: 2.41
 */

#ifdef SOM_Module_prappl_Source
#define SOM_Module_prappl_Source
#endif
#define PrintAppl_Class_Source

#ifdef _MVS
#include <DD:IH(exp4appl)>
#else
#include "exp4appl.i.h"
#endif

SOM_Scope void SOMLINK somDefaultInit(PrintAppl somSelf, somInitCtrl* ctrl)
{
    PrintApplData *somThis; /* set in BeginInitializer */
    somInitCtrl globalCtrl;
    somBooleanVector myMask;
    PrintApplMethodDebug("PrintAppl","somDefaultInit");
    PrintAppl_BeginInitializer_somDefaultInit;

    PrintAppl_Init_ISApplicator_somDefaultInit(somSelf, ctrl);

    /*
     Just overridden for performance reasons
     */
}

SOM_Scope void SOMLINK somDestruct(PrintAppl somSelf, octet doFree,
                                   somDestructCtrl* ctrl)
{
    PrintApplData *somThis; /* set in BeginDestructor */
    somDestructCtrl globalCtrl;
    somBooleanVector myMask;
    PrintApplMethodDebug("PrintAppl","somDestruct");
    PrintAppl_BeginDestructor;

    /*
     Just overridden for performance reasons
     */

    PrintAppl_EndDestructor;
}

SOM_Scope boolean SOMLINK applyTo(PrintAppl somSelf, Environment *ev,
                                   SOMObject element)
{
    /* PrintApplData *somThis = PrintApplGetData(somSelf); */
    PrintApplMethodDebug("PrintAppl","applyTo");

    StringObject_print_stringValue(element , ev ) ;

    return (TRUE) ;
}

```

```

}

----- exp4ini.c -----
#include <exp4ele.h>
#include <exp4appl.h>

#ifdef __IBMC__
#pragma linkage(SOMInitModule, system)
#endif

SOMEXTERN void SOMLINK SOMInitModule (long majorVersion,
                                       long minorVersion,
                                       string className)
{
    StringObjectNewClass(StringObject_MajorVersion , StringObject_MinorVersion);
    PrintApplNewClass(PrintAppl_MajorVersion , PrintAppl_MinorVersion);
    return;
}
----- exp4main.c -----
#include <stdio.h>
#include <assert.h>

#include <som.h>

#include <ssglobal.h>

#include <sstk.h>
#include <exp4ele.h>
#include <exp4appl.h>

#define _OK ; assert(ev->_major == NO_EXCEPTION)

#define _ERR ; assert(ev->_major == USER_EXCEPTION); \
    ev->_major = NO_EXCEPTION; \
    somPrintf("\nexception: %s\n",somExceptionId(ev));

extern int SOM_TraceLevel ;

int main (int argc)
{

    Environment *   ev ;

    ISStack        stack1 , stack2 ;
    StringObject    aStringObject ;
    string          aString ;
    PrintAppl       printAppl ;

    somPrintf("\nPushPop Sample running ... \n");

    SOM_TraceLevel = 0 ;
    if(argc==2) SOM_TraceLevel = 1 ;

    ev              = (Environment *) somGetGlobalEnvironment() ;
                   SOM_InitEnvironment(ev) ;

    printAppl       = (PrintAppl) PrintApplNew() ;

    stack1          = ISStackNew_ISStack_withOps( ev , aStringOps ) ; _OK ;
    stack2          = ISStackNew_ISStack_withOps( ev , aStringOps ) ; _OK ;

    aStringObject   = StringObjectNew_StringObject_withString(ev, "The" ) ; _OK ;
    _push(stack1 , ev, aStringObject ) ; _OK ;
    aStringObject   = StringObjectNew_StringObject_withString(ev , "quick" ) ; _OK ;

```

```

_push(stack1 , ev, aStringObject ) ; _OK ;
aStringObject = StringObjectNew_StringObject_withString(ev ,"brown" ) ; _OK ;
_push(stack1 , ev, aStringObject ) ; _OK ;
aStringObject = StringObjectNew_StringObject_withString(ev, "fox" ) ; _OK ;
_push(stack1 , ev, aStringObject ) ; _OK ;
aStringObject = StringObjectNew_StringObject_withString(ev , "jumps" ) ; _OK ;
_push(stack1 , ev, aStringObject ) ; _OK ;
aStringObject = StringObjectNew_StringObject_withString(ev ,"over" ) ; _OK ;
_push(stack1 , ev, aStringObject ) ; _OK ;
aStringObject = StringObjectNew_StringObject_withString(ev ,"a" ) ; _OK ;
_push(stack1 , ev, aStringObject ) ; _OK ;
aStringObject = StringObjectNew_StringObject_withString(ev ,"lazy" ) ; _OK ;
_push(stack1 , ev, aStringObject ) ; _OK ;
aStringObject = StringObjectNew_StringObject_withString(ev , "dog" ) ; _OK ;
_push(stack1 , ev, aStringObject ) ; _OK ;

somPrintf ("Content of Stack1 : \n") ;

_allElementsDo(stack1, ev, printAppl ) ; _OK ;

somPrintf ("\n") ;
somPrintf ("-----\n") ;
somPrintf ("\n") ;

while ( ! _isEmpty (stack1, ev ) )
{
    _popWithElement(stack1, ev , &aStringObject) ; _OK ;
    _push(stack2, ev , aStringObject) ; _OK ;
}

somPrintf ("Content of Stack2 : \n") ;

_allElementsDo(stack2, ev, printAppl ) ; _OK ;

somPrintf ("\n") ;
somPrintf ("-----\n") ;
somPrintf ("\n") ;

while ( ! _isEmpty (stack2, ev ) )
{
    _popWithElement(stack2, ev , &aStringObject) ; _OK ;
    somPrintf("Popped from stack2 : " ) ;
    _print_stringValue(aStringObject, ev) ; _OK ;
    somPrintf("\n" ) ;
}

return (0) ;

}

```

This program produces the following output:

PushPop Sample running ...
Content of Stack1 :

The
quick
brown
fox
jumps
over
a
lazy
dog

Content of Stack2 :

dog
lazy
a
over
jumps
fox
brown
quick
The

Popped from stack2 :The
Popped from stack2 :quick
Popped from stack2 :brown
Popped from stack2 :fox
Popped from stack2 :jumps
Popped from stack2 :over
Popped from stack2 :a
Popped from stack2 :lazy
Popped from stack2 :dog

Index

A

- abstract Collection Classes
 - equality collection 131
 - equality sorted collection 145
 - key collection
 - restriction on replacing elements 29
 - sorted collection 137
- accessing elements 22, 31
- addAsFirst() function 28
- addAsLast() function 28
- addAsNext() function 28
- addAsPrevious() function 28
- adding elements to collections
 - effect on cursors 30
 - overview 28
- addOrReplaceElementWithKey() function 23
- allElementsDo() function 32
- IApplicatorOverrideException 40
- applyTo() function
 - applicator classes 119
- assign 54
- Assign() function
 - operations class 125
- auxiliary class 17, 24

B

- Bag 73—74
 - Deque 13
 - description 13
 - properties of 19
- IBoolean 45
- bounded collections 33

C

- C++ SOM and Cross-language SOM Class Libraries class
 - general types of Collection Classes 17
 - hierarchy
 - abstract collections 25
 - Collection Class Library 26
- CLASS_BASE_NAME 45
- CLASS_NAME 45
- collection
 - conditions for equality 58
 - copying 28
 - creating 28
 - cursor association 30
 - iterating over 31
 - modifying 28—30
 - using polymorphism with 37

- collection abstract class 129
- Collection Class Library
 - abstract classes
 - collection 129
 - equality collection 131
 - equality key collection 141
 - equality key sorted collection 147
 - equality sorted collection 145
 - key collection 133
 - key sorted collection 143
 - ordered collection 135
 - sequential collection 139
 - sorted collection 137
 - applicator class 119, 121
 - concrete classes
 - Bag collection 73
 - Deque collection 75
 - Equality Sequence c collection 77
 - Heap collection 79
 - Key Bag collection 81
 - Key Set collection 83
 - Key Sorted Bag collection 85
 - Key Sorted Set collection 87
 - Map collection 89
 - Priority Queue collection 91
 - Queue collection 93
 - Relation collection 95
 - Sequence collection 97
 - Set collection 99
 - Sorted Bag collection 101
 - Sorted Map collection 103
 - Sorted relation collection 105
 - Sorted Set collection 107
 - Stack collection 109
 - cursor classes 115
 - Global 113
 - operations class 125
 - predicate class 123
 - reasons for using 17
 - steps for using 27
 - types of collections 17
- IComparatorOverrideException 40
- compare() function
 - Collection Class Library 54
 - using separate functions 35
- comparator classes 121
- operations class 125
- concrete implementations 25
- constant iterator class 32
- constructors
 - cursor classes 115

- containment function 23
- copy() function
 - flat collections 55
- creating a collection 28
- ICursorInvalidException 40
- cursors
 - accessing elements with 31
 - association with a collection 30
 - description 30—31
 - effect of removing elements 29
 - effect of replacing elements 29
 - iteration 32
 - locating elements with 31
 - properties that may cause an exception 40
 - reasons for using 30
 - removing elements with 29
 - validity 30

D

- Deque 24, 75
- destructor, flat collections 47
- destructors
 - flat collections 47
- difference
 - definition for Bags 73
 - definition for flat collections 56

E

- element equality 18, 20—22
- element() function
 - cursor classes 116
- elementAt() function
 - accessing elements with 31
 - flat collection classes 56
 - replacing elements using 31
 - role in Collection Class Library 29
- elements in Collection Class Library
 - accessing 22, 31
 - adding 23, 28
 - effect on cursors 30
 - functions
 - introduction 35
 - methods for providing 35
 - using element operation classes 36
 - iterating 31
 - locating 23, 31
 - See also* locate... functions
 - occurrence 29
 - operation classes 36
 - polymorphism 37
 - removing 29
 - effect on cursors 30
 - replacing 29—30
 - using elementAt() function 31

- elements in Collection Class Library (*continued*)
 - value 29
- elementWithKey() function 31
- EmptyException 41
- equal 58
- equal element 45
- Equal() function
 - operations class 125
- equality collection abstract class 131
- equality key collection abstract class 141
- equality key sorted collection abstract class 147
- equality relation 20
- Equality Sequence 13, 19, 77—78
- equality sorted collection abstract class 145
- equality test
 - in Collection Class Library
 - using separate functions 35
- evaluateFor() function
 - predicate classes 123
- examples
 - machine-readable xx
 - naming of xx
 - softcopy xx
- exception
 - IApplicatorOverrideException 40
 - IComparatorOverrideException 40
 - ICursorInvalidException 40
 - EmptyException 41
 - Exception 42
 - IFullException 33, 41
 - IdenticalCollectionException 41
 - in Collection Class Library 39
 - INoSOMObjectException
 - InvalidObjectException 41
 - InvalidReplacementException 41
 - IKeyAlreadyExistsException 41
 - INotBoundedException 33, 41
 - INotContainsKeyException 41
 - OpsInUseException
 - OpsInvalidException 41
 - IPositionInvalidException 42
 - IPredicateOverrideException 42
 - IRemoteCollectionException 42
 - SOMObjectException 42
 - IUserApplicatorException 42
 - IUserComparatorException
 - IUserOpsException 42
 - IUserPredicateException 42
 - violated precondition in Collection Class Library 39

F

- firstElement() function 31
- flat collection classes
 - overview 17—23
 - with restricted access 24

IFullException 41

H

Hash() function

operations class 126

hashing

restriction on replacing elements 29

header files

See chapters on individual classes

Heap 79—80

description 14

properties of 19

replacing elements 29

hierarchy

See class — hierarchy

I

IOutOfCollectionMemoryException

IdenticalCollectionException 41

IDL xiii

implementation in Collection Class Library

concrete 25

initializer methods

Collection Class Library

flat collections 47

initializer methods, flat collections 47

INoSOMObjectException

Interface Definition Language xiii

intersection

Bags 73

flat collections 58

invalidate() function

cursor classes 116

InvalidObjectException 41

InvalidReplacementException 41

isEmpty() function

flat collections 58

isValid() function

cursor classes 116

limitation 30

role of 40

iteration

over collections 31

restrictions 31

iteration order 45

iterator class 32

K

key access

basic properties of flat collections 18

description 19

overview 19—23

Key Bag 14, 19, 23, 81—82

key collection 29

restriction on replacing elements 29

key collection abstract class 133

key equality 20—22

Key Set 83—84

adding elements 23

description 14

properties of 19

Key Sorted Bag 14, 19, 85—86

key sorted collection abstract class 143

Key Sorted Set 14, 19, 87—88

key-type functions

introduction 35

methods for providing 35

using element operation classes 36

Key() function

operations class 126

KeyAlreadyExistsException 41

KeyCompare() function

operations class 126

KeyEqual() function

operations class 126

KeyHash() function

operations class 126

L

last-in, first-out behavior (LIFO) 109

LIFO (last-in, first-out) behavior 109

linked implementation 30

locateOrAddElementWithKey() function 23

locating elements 31

M

Map 15, 89—90

memory management

using separate functions 35

modifying a collection 28—30

multiple collections 18, 22—23

N

newCursor() function

flat collections 63

newElementCursor() function

flat collections 63

newOrderedCursor() function

flat collections 64

NotBoundedException 41

NotContainsKeyException 41

notEqual() 64

Number Collection Class type 45

numberOfElements() function

flat collections 64

O

- operations classes
 - using 36
- operator !=
 - cursor classes 116
 - flat collections 64
- operator =
 - flat collections 54
- operator ==
 - cursor classes 116
 - flat collections 58
- operator!= Collection Class Function 64
- operator= Collection Class Function 54
- operator== Collection Class Function 58
- OpsInUseException
- OpsInvalidException 41
- ordered collection
 - multiple inheritance 38
 - removing an element 29
- ordered collection abstract class 135
- ordering relation
 - as a collection property 18
 - possible orderings of collections 19
 - restriction on replacing elements 29
 - sorted collections 19
- OutOfMemory exception 41

P

- polymorphism
 - introduction 37
- IPosition Collection Class type 45
- position function 65
- positioning property 45
- IPositionInvalidException 42
- IPostorder Collection Class type 45
- precondition
 - violated 39—40
- predicate objects in Collection Class Library 29
- IPredicateOverrideException 42
- IPreorder Collection Class type 45
- Priority Queue 15, 91—92
 - Deque() 56
 - enqueue() 57

Q

- Queue 15
- queue collection 93—94
 - Deque() 56
 - enqueue() 57

R

- Relation 15, 95—96

- IRemoteCollectionException 42
- remove() function 65
 - Collection Class Library 65
 - behavior of 29
 - role of 28
- removeAll() function
 - flat collections 66
 - notes on using 29
- removeAllElementsWithKey() function 66
- removeAllOccurrences() function 66
- removeAt() function 66
- removeAtPosition() function 67
- removeElementWithKey() function 67
- removeFirst() function 29, 67
- removeLast() function 29, 67
- removing elements
 - See also* remove... functions for collections
 - effect on cursors 30
 - overview 29
- replace() function 28
- replaceAt() function
 - flat collections 68
 - role of 29
- replaceElementWithKey() function 68
- replacing elements 29—30
 - See also* replace... functions for collections
 - using elementAt() function 31

S

- same key 45
- Sequence 15, 97—98
- sequential collection abstract class 139
- sequential collections
 - add() behavior with 48
 - conditions for equality 58
 - replacing elements 29
- Set collection 99—100
- setToFirst() function
 - cursor classes 116
 - flat collections 68
- setToLast() function
 - cursor classes 117
 - flat collections 69
- setToNext() function
 - cursor classes 117
 - flat collections 69
- setToNextDifferentElement() function 69
- setToNextWithDifferentKey() function 70
- setToPosition() function 70
- setToPrevious() function
 - cursor classes 117
 - flat collections 70
- SOM-enabled and Not SOM-enabled Versions
 - C++ SOM and Cross-language SOM Collection Classes

- SOM-enabled and Not SOM-enabled Versions
 - (*continued*)
 - Compiling and Binding with the C++ SOM Libraries
 - Using the Cross-language SOM Collection Classes
- SOMObjectException 42
- sort() function 71
- Sorted Bag 16, 19, 101—102
- sorted collection abstract class 137
- sorted collections
 - add() behavior with 48
 - multiple inheritance 38
 - ordering relation 19
- Sorted Map 16, 103—104
 - description 16
 - properties of 19
 - restrictions for adding elements 23
 - Sorted Relation 16
- Sorted Relation 105—106
 - properties of 19
- Sorted Set 16, 19, 107—108
- Stack 17, 24, 109—110
- system failures 40
- system restrictions 40

T

- tabular implementation 30
- templates
- this collection 46
- top() function 71
- ITreeIterationOrder Collection Class type 45

U

- unbounded collections 33
- undefined cursor 45
- union
 - Bags 73
 - flat collections 71
- unionWith() function 71
- unique collections
 - add() behavior with 48
 - adding elements 28
 - compared to multiple collections 18
 - conditions for equality 58
 - description 22—23
- unordered collections
 - characteristics 19
 - cursor iteration drawbacks 32
 - locateNext() 61
 - locateNextElementWithKey() 61
- IUserApplicatorException 42
- IUserComparatorException 42
- IUserOpsException 42
- IUserPredicateException 42

Communicating Your Comments to IBM

OS/390

C/C++

SOM-Enabled Class Library User's Guide and Reference

Publication No. SC09-2366-02

If there is something you like—or dislike—about this book, please let us know. You can use one of the methods listed below to send your comments to IBM. If you want a reply, include your name, address, and telephone number. If you are communicating electronically, include the book title, publication number, page number, or topic you are commenting on.

The comments you send should only pertain to the information in this book and its presentation. To request additional publications or to ask questions or make comments about the functions of IBM products or systems, you should talk to your IBM representative or to your IBM authorized remarketer.

When you send comments to IBM, you grant IBM a nonexclusive right to use or distribute your comments in any way it believes appropriate without incurring any obligation to you.

If you are mailing a readers' comment form (RCF) from a country other than the United States, you can give it to the local IBM branch office or IBM representative for postage-paid mailing.

- If you prefer to send comments by mail, use the RCF at the back of this book.
- If you prefer to send comments by FAX, use this number:
 - United States and Canada: 416-448-6161
 - Other countries: (+1)-416-448-6161
- If you prefer to send comments electronically, use the network ID listed below. Be sure to include your entire network address if you wish a reply.
 - Internet: torrcf@ca.ibm.com
 - IBMLink: [toribm\(torrcf\)](mailto:toribm(torrcf)@ca.ibm.com)
 - IBM/PROFS: [torolab4\(torrcf\)](mailto:torolab4(torrcf)@ca.ibm.com)
 - IBMMAIL: [ibmmail\(caibmwt9\)](mailto:ibmmail(caibmwt9)@ca.ibm.com)

Readers' Comments — We'd Like to Hear from You

OS/390

C/C++

SOM-Enabled Class Library User's Guide and Reference

Publication No. SC09-2366-02

Overall, how satisfied are you with the information in this book?

	Very Satisfied	Satisfied	Neutral	Dissatisfied	Very Dissatisfied
Overall satisfaction	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

How satisfied are you that the information in this book is:

	Very Satisfied	Satisfied	Neutral	Dissatisfied	Very Dissatisfied
Accurate	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Complete	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Easy to find	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Easy to understand	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Well organized	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Applicable to your tasks	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

Please tell us how we can improve this book:

Thank you for your responses. May we contact you? ☐ Yes ☐ No

When you send comments to IBM, you grant IBM a nonexclusive right to use or distribute your comments in any way it believes appropriate without incurring any obligation to you.

Name

Address

Company or Organization

Phone No.



Fold and Tape

Please do not staple

Fold and Tape

PLACE
POSTAGE
STAMP
HERE

IBM Canada Ltd. Laboratory
Information Development
2G/345/1150/TOR
1150 EGLINTON AVENUE EAST
NORTH YORK ONTARIO CANADA M3C 1H7

Fold and Tape

Please do not staple

Fold and Tape



Printed in the United States of America
on recycled paper containing 10%
recovered post-consumer fiber.

SC09-2366-02

