SOMobjects Developer Toolkit

Users Guide

An introductory guide to the System Object Model and its accompanying frameworks

Version 2.1
October 1994

Note: Before using this information and the product it supports, be sure to read the trademark information under "Trademarks" on page xiii.

Version 2.1 (October 1994)

The following paragraph does not apply to the United Kingdom or any country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THE PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions; therefore, this statement may not apply to you.

This publication could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time.

It is possible that this publication may contain reference to, or information about, IBM products (machines and programs), programming, or services that are not announced in your country. Such references or information must not be construed to mean that IBM intends to announce such IBM products, programming, or services in your country.

Requests for technical information about IBM products should be made to your IBM Authorized Dealer or your IBM Marketing Representative.

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 Commercial Relations, IBM Corporation, Purchase, NY 10577.

COPYRIGHT LICENSE: This publication contains printed sample application programs in source language, which illustrate AIX, OS/2, or Windows programming techniques. You may copy and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the AIX, OS/2, or Windows application programming interface.

Each copy of any portion of these sample programs or any derivative work, which is distributed to others, must include a copyright notice as follows: "©(your company name) (year) All Rights Reserved."

However, the following copyright notice protects this documentation under the Copyright laws of the United States and other countries which prohibit such actions as, but not limited to, copying, distributing, modifying, and making derivative works.

© Copyright International Business Machines Corporation, 1991 — 1994. All rights reserved.

The term "IBM" is a registered trademark and "SOMobjects" and "System Object Model" are trademarks of International Business Machines Corporation.

Notice to US 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.

SOMobjects Developer Toolkit Users Guide

Contents

About This Book ..... **xi**

Trademarks ..... **xiii**

Chapter 1. Introduction to the SOMobjects Developer Toolkit

1.1 Background ..... **1 – 1**

1.2 Introducing SOM and the SOMobjects Toolkit ..... **1 – 3**

The SOM Compiler ..... 1 – 4

The SOM run-time library ..... 1 – 5

Frameworks provided in the SOMobjects Toolkit ..... 1 – 5

1.3 What's New in SOMobjects Version 2.1 ..... **1 – 7**

1.4 Overview of this book ..... **1 – 10**

Chapter 2. Tutorial for Implementing SOM Classes

2.1 Basic Concepts of the System Object Model (SOM) ..... **2 – 1**

Development of the Tutorial examples ..... 2 – 5

2.2 Basic Steps for Implementing SOM Classes ..... **2 – 6**

Example 1 — Implementing a Simple Class with One Method ..... **2 – 7**

Example 2 — Adding an Attribute to the Hello class ..... **2 – 13**

Attributes vs instance variables ..... 2 – 15

Example 3 — Overriding an Inherited Method ..... **2 – 17**

Example 4 — Initializing a SOM Object ..... **2 – 20**

Example 5 — Using Multiple Inheritance ..... **2 – 22**

Chapter 3. Using SOM Classes in Client Programs ..... **3 – 1**

3.1 An Example Client Program ..... **3 – 3**

3.2 Using SOM Classes — the Basics ..... **3 – 4**

Declaring object variables ..... 3 – 4

Creating instances of a class ..... 3 – 5

Invoking methods on objects ..... 3 – 8

Using class objects ..... 3 – 20

Compiling and linking ..... 3 – 23

3.3 Language-neutral Methods and Functions ..... **3 – 25**

Generating output ..... 3 – 25

Getting information about a class ..... 3 – 25

Getting information about an object ..... 3 – 27

Debugging ..... 3 – 28

Checking the validity of method calls ..... 3 – 29

Exceptions and error handling ..... 3 – 29

Memory management ..... 3 – 35

SOM manipulations using somId's ..... 3 – 36

Chapter 4. SOM IDL and the SOM Compiler

4.1 Interface vs Implementation ..... **4 – 1**

4.2 SOM Interface Definition Language ..... **4 – 3**

Include directives ..... 4 – 4

Type and constant declarations ..... 4 – 4

Exception declarations ..... 4 – 10

Interface declarations . . . . . 4 – 12
Constant, type, and exception declarations . . . . . 4 – 13
Attribute declarations . . . . . 4 – 14
Method (operation) declarations . . . . . 4 – 14
Implementation statements . . . . . 4 – 17
Comments within a SOM IDL file . . . . . 4 – 28
Designating 'private' methods and attributes . . . . . 4 – 29
Defining multiple interfaces in a .idl file . . . . . 4 – 29
Scoping and name resolution . . . . . 4 – 30
Extensions to CORBA IDL permitted by SOM IDL . . . . . 4 – 31
4.3 The SOM Compiler . . . . . 4 – 33
Generating binding files . . . . . 4 – 33
Environment variables affecting the SOM Compiler . . . . . 4 – 36
Running the SOM Compiler . . . . . 4 – 38
4.4 The 'pdl' Facility . . . . . 4 – 43

Chapter 5. Implementing Classes in SOM

5.1 The SOM Run-time Environment . . . . . 5 – 1
Run-time environment initialization . . . . . 5 – 1
Parent class vs. metaclass . . . . . 5 – 4
SOM-derived metaclasses . . . . . 5 – 7
5.2 Inheritance . . . . . 5 – 10
5.3 Method Resolution . . . . . 5 – 13
Offset resolution . . . . . 5 – 13
Name-lookup resolution . . . . . 5 – 13
Dispatch-function resolution . . . . . 5 – 14
Customizing method resolution . . . . . 5 – 14
The four kinds of SOM methods . . . . . 5 – 15
5.4 Implementing SOM Classes . . . . . 5 – 16
The implementation template . . . . . 5 – 17
Extending the implementation template . . . . . 5 – 19
Compiling and linking . . . . . 5 – 23
5.5 Initializing and Uninitializing Objects . . . . . 5 – 25
Initializer methods . . . . . 5 – 25
Uninitialization . . . . . 5 – 32
A complete example . . . . . 5 – 33
Customizing the initialization of class objects . . . . . 5 – 39
5.6 Creating a SOM Class Library . . . . . 5 – 40
General guidelines for class library designers . . . . . 5 – 40
Types of class libraries . . . . . 5 – 41
Building export files . . . . . 5 – 41
Specifying the initialization function . . . . . 5 – 43
Creating the import library . . . . . 5 – 44
5.7 Customizing Memory Management . . . . . 5 – 48
5.8 Customizing Class Loading and Unloading . . . . . 5 – 49
Customizing class initialization . . . . . 5 – 49
Customizing DLL loading . . . . . 5 – 49
Customizing DLL unloading . . . . . 5 – 50
5.9 Customizing Character Output . . . . . 5 – 51
5.10 Customizing Error Handling . . . . . 5 – 52
5.11 Customizing Mutual Exclusion Services (Thread Safety) . . . . . 5 – 53
5.12 Customizing Multi-threading Services . . . . . 5 – 55

Chapter 6. Distributed SOM (DSOM)

6.1 Introduction . . . . . 6 – 1
What is Distributed SOM? . . . . . 6 – 1
Chapter outline . . . . . 6 – 2
6.2 A Simple DSOM Example . . . . . 6 – 4
The “Stack” interface . . . . . 6 – 4
The “Stack” class implementation . . . . . 6 – 5
Client program using a local stack . . . . . 6 – 6
Client program using a remote stack . . . . . 6 – 8
A note on finding existing objects . . . . . 6 – 11
“Stack” server implementation . . . . . 6 – 12
Compiling the application . . . . . 6 – 12
Installing the implementation . . . . . 6 – 12
Running the application . . . . . 6 – 14
“Stack” example run-time scenario . . . . . 6 – 14
Summary . . . . . 6 – 16
6.3 Basic Client Programming . . . . . 6 – 17
DSOM Object Manager . . . . . 6 – 17
Initializing a client program . . . . . 6 – 18
Exiting a client program . . . . . 6 – 18
Creating remote objects . . . . . 6 – 19
Destroying remote objects . . . . . 6 – 22
Creating remote objects using user-defined metaclasses . . . . . 6 – 23
Saving and restoring references to objects . . . . . 6 – 24
Finding existing objects . . . . . 6 – 25
Invoking methods on remote objects . . . . . 6 – 26
Passing object references in method calls . . . . . 6 – 27
Memory management . . . . . 6 – 27
Writing clients that are also servers . . . . . 6 – 30
Compiling and linking clients . . . . . 6 – 30
6.4 Basic Server Programming . . . . . 6 – 31
Server run-time objects . . . . . 6 – 31
Server activation . . . . . 6 – 32
Initializing a server program . . . . . 6 – 33
Processing requests . . . . . 6 – 34
Exiting a server program . . . . . 6 – 35
Managing objects in the server . . . . . 6 – 35
Example: Writing a persistent object server . . . . . 6 – 39
Identifying the source of a request . . . . . 6 – 42
Compiling and linking servers . . . . . 6 – 43
6.5 Implementing Classes . . . . . 6 – 44
Using SOM class libraries . . . . . 6 – 44
Using other object implementations . . . . . 6 – 46
Parameter memory management . . . . . 6 – 48
Building and registering class libraries . . . . . 6 – 49
6.6 Configuring DSOM Applications . . . . . 6 – 50
Preparing the environment . . . . . 6 – 50
Registering class interfaces . . . . . 6 – 51
Registering servers and classes . . . . . 6 – 51
Verifying the DSOM environment with ‘somdchk’ . . . . . 6 – 60
Freeing interprocess communication resources on AIX . . . . . 6 – 62

6.7 Running DSOM Applications ..... 6 – 63
6.8 DSOM as a CORBA-compliant Object Request Broker ..... 6 – 64
Mapping OMG CORBA terminology onto DSOM ..... 6 – 64
6.9 Advanced Topics ..... 6 – 72
Peer vs. client-server processes ..... 6 – 72
Dynamic Invocation Interface ..... 6 – 75
Building client-only “stub” DLLs ..... 6 – 81
Creating user-supplied proxies ..... 6 – 81
Customizing the default base proxy class ..... 6 – 83
Sockets class ..... 6 – 84
6.10 Error Reporting and Troubleshooting ..... 6 – 85
Troubleshooting hints ..... 6 – 86
6.11 Limitations ..... 6 – 90

Chapter 7. The Interface Repository Framework

7.1 Introduction ..... 7 – 1
7.2 Using the SOM Compiler to Build an Interface Repository ..... 7 – 2
7.3 Managing Interface Repository files ..... 7 – 3
The SOM IR file “som.ir” ..... 7 – 3
Managing IRs via the SOMIR environment variable ..... 7 – 3
Placing ‘private’ information in the Interface Repository ..... 7 – 5
7.4 Programming with the Interface Repository Objects ..... 7 – 6
Methods introduced by Interface Repository classes ..... 7 – 7
Accessing objects in the Interface Repository ..... 7 – 8
A word about memory management ..... 7 – 10
Using TypeCode pseudo-objects ..... 7 – 11

Chapter 8. The Persistence Framework

8.1 Introduction ..... 8 – 1
8.2 The Telephone-Directory Application ..... 8 – 1
Example 1: Nonpersistent telephone-directory example ..... 8 – 2
8.3 Persistent Objects ..... 8 – 7
Example 2: Single inheritance definition of persistent “phoneDir” ..... 8 – 8
Implementation of the persistent telephone directory ..... 8 – 9
Example 3: Definition of persistent “dirEntry” ..... 8 – 12
Example 4: Multiple inheritance definition of persistent “pphoneDir” ..... 8 – 12
Embedded objects ..... 8 – 14
Persistent object IDs ..... 8 – 15
I/O groups ..... 8 – 16
8.4 Saving and Restoring Persistent Objects ..... 8 – 18
Saving a persistent object ..... 8 – 18
Restoring a persistent object ..... 8 – 18
Example 5: Storing and restoring a persistent “phoneDir” ..... 8 – 19
Persistent Object ID initialization ..... 8 – 23
Example 6: Storing objects in multiple files using system-assigned IDs ..... 8 – 26
Read/Write without children ..... 8 – 27
Choosing I/O Group Manager SOMPAscii or SOMPBinary ..... 8 – 30
SOMPAscii and SOMPBinary characteristics ..... 8 – 30
Activation and passivation ..... 8 – 35

8.5 Managing Persistent Objects ..... **8 – 36**
Checking persistent object existence ..... 8 – 36
Deleting persistent objects ..... 8 – 36
Persistent object states ..... 8 – 36
Garbage collection ..... 8 – 37
8.6 Storing Objects in Specialized Formats. ..... **8 – 39**
Persistent object format ..... 8 – 39
Encoder/Decoders ..... 8 – 40
The default Encoder/Decoder ..... 8 – 41
Writing an Encoder/Decoder ..... 8 – 42
Methods supporting encoder/decoders ..... 8 – 42
Example 7: Encoder/Decoder example implementation ..... 8 – 45
8.7 Multi-thread Considerations ..... **8 – 51**
8.8 Error Handling ..... **8 – 51**

Chapter 9. The Replication Framework
9.1 Introduction ..... **9 – 1**
9.2 Principles of the Replication Framework ..... **9 – 2**
Steps in using Replication ..... 9 – 3
9.3 Components of the Framework ..... **9 – 4**
9.4 Making a "replicated" Class that provides Operation Logging ..... **9 – 5**
9.5 Making a "replicated" Class that provides Value Logging ..... **9 – 8**
9.6 Handling Directives ..... **9 – 10**
9.7 Repdraw: A Complete Example ..... **9 – 12**
Main program for "repdraw" (first part) ..... 9 – 13
IDL specification of the "tablet" class ..... 9 – 15
Implementation of the "tablet" class (first part) ..... 9 – 16
9.8 Miscellaneous Considerations ..... **9 – 18**
9.9 Limitations ..... **9 – 25**

Chapter 10. The Metaclass Framework ..... **10 – 1**
10.1 Framework Metaclasses for "Before/After" Behavior ..... **10 – 3**
The 'SOMMBeforeAfter' metaclass ..... 10 – 3
Composition of before/after metaclasses ..... 10 – 5
10.2 The 'SOMMSingleInstance' Metaclass ..... **10 – 8**
10.3 The 'SOMMTraced' Metaclass ..... **10 – 9**
10.4 The 'SOMRReplicable' Metaclass ..... **10 – 11**
10.5 Error Codes ..... **10 – 14**

Chapter 11. Collection Classes
11.1 Categories of Collection Classes ..... **11 – 1**
IsSame vs. IsEqual comparisons ..... 11 – 1
Class inheritance vs. element inheritance ..... 11 – 2
Object-initializer methods ..... 11 – 2
Naming conventions ..... 11 – 2
11.2 Abstract Classes ..... **11 – 3**
11.3 Main Collection Classes ..... **11 – 4**
Hash table class — somf_THashTable ..... 11 – 4
Dictionary class — somf_TDictionary ..... 11 – 5
Set class — somf_TSet ..... 11 – 5
Deque, queue, and stack class — somf_TDeque ..... 11 – 6
Linked list class — somf_TPrimitiveLinkedList ..... 11 – 6
Sorted sequence class — somf_TSortedSequence ..... 11 – 6
Priority queue class — somf_TPriorityQueue ..... 11 – 7
Choosing the best class ..... 11 – 7

11.4 Iterator Classes ..... **11 – 9**
11.5 Mixin Classes ..... **11 – 10**
11.6 Supporting Classes ..... **11 – 11**
11.7 Inheritance Hierarchy of the Collection Classes ..... **11 – 12**
11.8 Utility Collection Classes by Category ..... **11 – 13**

Chapter 12. The Event Management Framework

12.1 Event Management Basics ..... **12 – 1**
Model of EMan usage ..... 12 – 1
Event types ..... 12 – 1
Registration ..... 12 – 2
Unregistering for events ..... 12 – 4
An example callback procedure ..... 12 – 4
Generating client events ..... 12 – 4
Examples of using other events ..... 12 – 5
Processing events ..... 12 – 5
Interactive applications ..... 12 – 6
12.2 Event Manager Advanced Topics ..... **12 – 7**
Threads and thread safety ..... 12 – 7
Writing an X or MOTIF application ..... 12 – 7
Extending EMan ..... 12 – 7
Using EMan from C++ ..... 12 – 8
Using EMan from other languages ..... 12 – 8
Tips on using EMan ..... 12 – 8
12.3 Limitations ..... **12 – 9**
Use of EMan DLL ..... 12 – 9

Appendix A. Customer Support and Error Codes **A–1**

Appendix B. Converting OIDL Files to IDL **B–1**

Appendix C. SOM IDL Language Grammar **C–1**

Appendix D. Subclassing the Persistence Framework

Persistence Framework Class Interaction ..... **D – 1**
Choosing Which Classes to Subclass ..... **D – 2**
Implementing New Persistence Framework Classes ..... **D – 4**
Implementing a new Encoder/Decoder ..... D – 4
Implementing a new or enhanced Media Interface ..... D – 6
Implementing a new I/O Group Manager ..... D – 12
Persistent Storage Manager interaction ..... D – 16
Template for an I/O Group Manager ..... D – 18
An Example I/O Group Manager and Media Interface Implementation ..... **D – 25**
The Media Interface ..... D – 25
The I/O Group Manager ..... D – 34

Appendix E. Implementing Sockets Subclasses

Sockets IDL interface ..... E – 1
IDL for a Sockets subclass ..... E – 5
Implementation considerations ..... E – 7
Example code ..... E – 7

Appendix F. emitcom: An Emitter of COM Interfaces
'emitcom' Syntax . . . . . F – 1
Execution of 'emitcom' . . . . . F – 1
Interface Identifiers . . . . . F – 2
User Procedure . . . . . F – 2
The Generated Interface . . . . . F – 4
Customizing the <comstem>.mak . . . . . F – 4
Example . . . . . F – 4
Limitations . . . . . F – 7

Glossary . . . . . Glos – 1

Index . . . . . Index – 1

About This Book

This book describes the **System Object Model** (**SOM**) of the **SOMobjects Developer Toolkit** and explains how programmers using C, C++, and other languages can:

- Implement class libraries that exploit the SOM library-packaging technology,
- Develop client programs that use class libraries that were built using SOM, and
- Develop applications that use the *frameworks* supplied with the SOMobjects Toolkit, class libraries that facilitate development of object-oriented applications.

In addition to this book, refer to the *SOMobjects Developer Toolkit Programmers Reference Manual* during application development for specific information about the classes, methods, functions, and macros supplied with the SOMobjects Toolkit. Also, refer to the *SOMobjects Developer Toolkit: Emitter Framework Guide and Reference* for documentation of the Emitter Framework of the SOMobjects Toolkit. In addition, the *SOMobjects Developer Toolkit: Collection Classes Reference Manual* describes the collection classes and methods provided with the SOMobjects Toolkit.

How This Book Is Organized

This book contains twelve chapters, five appendices, a glossary, and an index.

- The first part (chapters 1 and 2) introduces the reader to the SOMobjects Toolkit, gives an overview of the major elements of SOM, and provides a Tutorial containing several evolutionary examples.
- The second part (chapter 3) describes how to develop client programs in C, C++, or other languages to use classes that were implemented using SOM, including how to create instances of a class and call methods on them.
- The third part (chapters 4 and 5) gives the syntax of the SOM Interface Definition Language, provides directions for running the SOM Compiler, describes the SOM run-time environment, and presents advanced information about SOM's object model and how to customize SOM for particular applications.
- The fourth part (chapters 6 through 12) contains information about the frameworks composing the SOMobjects Toolkit: Distributed SOM (DSOM), the Interface Repository Framework, the Persistence Framework, the Replication Framework, the Metaclass Framework, and the Event Management Framework. (The Emitter framework is documented separately, in the *Emitter Framework Guide and Reference*.) This part also describes the utility metaclasses provided with the SOMobjects Toolkit, as well as a large set of Collection Classes provided as a convenience to programmers.
- The appendices describe customer support-procedures and error codes, give directions for converting existing files from the SOM Version 1 OIDL syntax to the current IDL syntax, and provide the grammar for SOM IDL. Also included is a thorough discussion of subclassing the Persistence Framework, complete with extensive examples. Another appendix describes how to subclass a "Sockets" class that facilitates inter-process communications required by some of the frameworks. Lastly is a description of the 'emitcom' program, which creates bindings that allow SOM classes to be used with COM (Microsoft's component interface model).
- The Glossary provides brief definitions of terminology related to SOM and the SOMobjects Toolkit. Finally, an extensive Index enables the reader to locate specific information quickly.

Who Should Read This Book

This book is for the professional programmer using C, C++, or another language who wishes to:

- Use SOM to build object–oriented class libraries, or
- Write application programs using class libraries that others have implemented using SOM,

even if the programming language does not directly support object–oriented programming.

The discussions in this book are expressed in the commonly used terminology of object-oriented programming. A number of important terms are everyday English words that take on specialized meanings. These terms appear in the Glossary at the back of this book. You may find it worth consulting the Glossary if the unusual significance attached to an otherwise ordinary word puzzles you.

This book assumes that you are an experienced programmer and that you have a general familiarity with the basic notions of object-oriented programming. Practical experience using an object-oriented programming language is helpful, but not essential.

If you would like a good introduction to object-oriented programming or a general survey of the many aspects of the topic, you might enjoy reading one of the following books:

- Booch, G, *Object-Oriented Design with Applications*, Benjamin/Cummings 1991, ISBN 0-8053-0091-0.
- Budd, T, *An Introduction to Object-Oriented Programming*, Addison-Wesley 1991, ISBN 0-201-54709-0.
- Cox, B, and Novobilski, A, *Object-Oriented Programming, An Evolutionary Approach* 2nd Edition, Addison-Wesley 1991, ISBN 0-201-54834-8.

Trademarks

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

AIX
IBM
Operating System/2
OS/2
OS/2 Workplace Shell
RISC System 6000
SOMobjects
System Object Model

For convenience, the acronym “SOM” is used in this publication to reference the technology of the System Object Model, and the term “SOM Compiler” is used to reference the compiler of the System Object Model.

Each of the following terms used in this publication is a trademark of another company:

Intel	Intel Corporation
IPX	Novell Corporation
Lotus 1-2-3	Lotus Development Corporation
Microsoft EXCEL	Microsoft Corporation
Microsoft Windows	Microsoft Corporation
NetWare	Novell Corporation
Objective-C	The Stepstone Corporation
Smalltalk	Digitalk Inc.

The term “ANSI C” used throughout this publication refers to American National Standard X3.159–1989.

The term “CORBA” used throughout this publication refers to the Common Object Request Broker Architecture standards promulgated by the Object Management Group, Inc.

Chapter 1. Introduction to the SOMobjects Toolkit

Contents

1.1 Background ..... 1 – 1

1.2 Introducing SOM and the SOMobjects Toolkit ..... 1 – 3

The SOM Compiler ..... 1 – 4

The SOM run-time library ..... 1 – 5

Frameworks provided in the SOMobjects Toolkit ..... 1 – 5

- Distributed SOM ..... 1 – 5
- Interface Repository Framework ..... 1 – 5
- Persistence Framework ..... 1 – 6
- Replication Framework ..... 1 – 6
- Emitter Framework ..... 1 – 6
- Metaclass Framework ..... 1 – 6

1.3 What's New in SOMobjects Version 2.1 ..... 1 – 7

1.4 Overview of this book ..... 1 – 10

Chapter 1. Introduction to the SOMobjects Toolkit

1.1 Background

Object-oriented programming (or **OOP**) is an important new programming technology that offers expanded opportunities for software reuse and extensibility. Object-oriented programming shifts the emphasis of software development away from functional decomposition and toward the recognition of units (called *objects*) that encapsulate both code and data. As a result, programs become easier to maintain and enhance. Object-oriented programs are typically more impervious to the "ripple effects" of subsequent design changes than their non-object-oriented counterparts. This, in turn, leads to improvements in programmer productivity.

Despite its promise, penetration of object-oriented technology to major commercial software products has progressed slowly because of certain obstacles. This is particularly true of products that offer only a binary programming interface to their internal object classes *(i.e.,* products that do not allow access to source code).

The first obstacle that developers must confront is the choice of an object-oriented programming language.

So-called "pure" object-oriented languages (such as Smalltalk) presume a complete run-time environment (sometimes known as a virtual machine), because their semantics represent a major departure from traditional, procedure-oriented system architectures. So long as the developer works within the supplied environment, everything works smoothly and consistently. When the need arises to interact with foreign environments, however (for example, to make an external procedure call), the pure-object paradigm ends, and objects must be reduced to data structures for external manipulation. Unfortunately, data structures do not retain the advantages that objects offer with regard to encapsulation and code reuse.

"Hybrid" languages such as C++, on the other hand, require less run-time support, but sometimes result in tight bindings between programs that implement objects (called "class libraries") and their clients (the programs that use them). That is, implementation detail is often unavoidably compiled into the client programs. Tight binding between class libraries and their clients means that client programs often must be recompiled whenever simple changes are made in the library. Furthermore, no binary standard exists for C++ objects, so the C++ class libraries produced by one C++ compiler cannot (in general) be used from C++ programs built with a different C++ compiler.

The second obstacle developers of object-oriented software must confront is that, because different object-oriented languages and toolkits embrace incompatible models of what objects are and how they work, software developed using a particular language or toolkit is naturally limited in scope. Classes implemented in one language cannot be readily used from another. A C++ programmer, for example, cannot easily use classes developed in Smalltalk, nor can a Smalltalk programmer make effective use of C++ classes. Object-oriented language and toolkit boundaries become, in effect, barriers to interoperability.

Ironically, no such barrier exists for ordinary procedure libraries. Software developers routinely construct procedure libraries that can be shared across a variety of languages, by adhering to standard linkage conventions. Object-oriented class libraries are inherently different in that no binary standards or conventions exist to derive a new class from an existing one, or even to invoke a method in a standard way. Procedure libraries also enjoy the benefit that their implementations can be freely changed without requiring client programs to be recompiled, unlike the situation for C++ class libraries.

For developers who need to provide binary class libraries, these are serious obstacles. In an era of open systems and heterogeneous networking, a single-language solution is frequently not broad enough. Certainly, mandating a specific compiler from a specific vendor in order to use a class library might be grounds not to include the class library with an operating system or other general-purpose product.

The **System Object Model (SOM)** is IBM's solution to these problems.

1.2 Introducing SOM and the SOMobjects Toolkit

The System Object Model (SOM) is a new object-oriented programming technology for building, packaging, and manipulating binary *class libraries*.

- With SOM, class implementors describe the *interface* for a class of objects (names of the methods it supports, the return types, parameter types, and so forth) in a standard language called the **Interface Definition Language**, or **IDL**.
- They then *implement* methods in their preferred programming language (which may be either an object-oriented programming language or a procedural language such as C).

This means that programmers can begin using SOM quickly, and also extends the advantages of OOP to programmers who use non-object-oriented programming languages.

A principal benefit of using SOM is that SOM accommodates changes in implementation details and even in certain facets of a class's interface, without breaking the binary interface to a class library and without requiring recompilation of client programs. As a rule of thumb, if changes to a SOM class do not require source-code changes in client programs, then those client programs will not need to be recompiled. This is not true of many object-oriented languages, and it is one of the chief benefits of using SOM. For instance, SOM classes can undergo structural changes such as the following, yet retain full backward, binary compatibility:

- Adding new methods,
- Changing the size of an object by adding or deleting instance variables,
- Inserting new parent (base) classes above a class in the inheritance hierarchy, and
- Relocating methods upward in the class hierarchy.

In short, implementors can make the typical kinds of changes to an implementation and its interfaces that evolving software systems experience over time.

Unlike the object models found in formal object-oriented programming languages, SOM is language-neutral. It preserves the key OOP characteristics of encapsulation, inheritance, and polymorphism, without requiring that the user of a SOM class and the implementor of a SOM class use the same programming language. SOM is said to be *language-neutral* for four reasons:

1. All SOM interactions consist of standard procedure calls. On systems that have a standard linkage convention for system calls, SOM interactions conform to those conventions. Thus, most programming languages that can make external procedure calls can use SOM.
2. The form of the SOM Application Programming Interface, or API (the way that programmers invoke methods, create objects, and so on) can vary widely from language to language, as a benefit of the SOM binding*s*. *Bindings* are a set of macros and procedure calls that make implementing and using SOM classes more convenient by tailoring the interface to a particular programming language.
3. SOM supports several mechanisms for method resolution that can be readily mapped into the semantics of a wide range of object-oriented programming languages. Thus, SOM class libraries can be shared across object-oriented languages that have differing object models. A SOM object can potentially be accessed with three different forms of method resolution:
   - Offset resolution: roughly equivalent to the C++ "virtual function" concept. Offset resolution implies a static scheme for typing objects, with polymorphism based strictly on class derivation. It offers the best performance characteristics for SOM method resolution. Methods accessible through offset resolution are called *static* methods, because they are considered a fixed aspect of an object's interface.

- Name-lookup resolution: similar to that employed by Objective-C and Smalltalk. Name resolution supports untyped (sometimes called "dynamically" typed) access to objects, with polymorphism based on the actual protocols that objects honor. Name resolution offers the opportunity to write code to manipulate objects with little or no awareness of the type or shape of the object when the code is compiled.
- Dispatch-function resolution: a unique feature of SOM that permits method resolution based on arbitrary rules known only in the domain of the receiving object. Languages that require special entry or exit sequences or local objects that represent distributed object domains are good candidates for using dispatch-function resolution. This technique offers the highest degree of encapsulation for the implementation of an object, with some cost in performance.

4. SOM conforms fully with the Object Management Group's (OMG) Common Object Request Broker Architecture (CORBA) standards.† In particular,
   - Interfaces to SOM classes are described in CORBA's Interface Definition Language, IDL, and the entire SOMobjects Toolkit supports all CORBA-defined data types.
   - The SOM bindings for the C language are compatible with the C bindings prescribed by CORBA.
   - All information about the interface to a SOM class is available at run time through a CORBA-defined "Interface Repository."

SOM is not intended to replace existing object-oriented languages. Rather, it is intended to complement them so that application programs written in different programming languages can share common SOM class libraries. For example, SOM can be used with C++ to

- Provide upwardly compatible class libraries, so that when a new version of a SOM class is released, client code needn't be recompiled, so long as no changes to the client's source code are required.
- Allow other language users (and other C++ compiler users) to use SOM classes implemented in C++.
- Allow C++ programs to use SOM classes implemented using other languages.
- Allow other language users to implement SOM classes derived from SOM classes implemented in C++.
- Allow C++ programmers to implement SOM classes derived from SOM classes implemented using other languages.
- Allow encapsulation (implementation hiding) so that SOM class libraries can be shared without exposing private instance variables and methods.
- Allow dynamic (run-time) method resolution in addition to static (compile-time) method resolution (on SOM objects).
- Allow information about classes to be obtained and updated at run time. (C++ classes are compile-time structures that have no properties at run time.)

The SOM Compiler

The **SOMobjects Toolkit** contains a tool, called the **SOM Compiler**, that helps implementors build classes in which *interface* and *implementation* are decoupled. The SOM Compiler reads the IDL definition of a class interface and generates:

- an implementation skeleton for the class,
- bindings for implementors, and
- bindings for client programs.

†OMG is an industry consortium founded to advance the use of object technology in distributed, heterogeneous environments.

Bindings are language-specific macros and procedures that make implementing and using SOM classes more convenient. These bindings offer a convenient interface to SOM that is tailored to a particular programming language. For instance, C programmers can invoke methods in the same way they make ordinary procedure calls. The C++ bindings "wrap" SOM objects as C++ objects, so that C++ programmers can invoke methods on SOM objects in the same way they invoke methods on C++ objects. In addition, SOM objects receive full C++ typechecking, just as C++ objects do. Currently, the SOM Compiler can generate both C and C++ language bindings for a class. The C and C++ bindings will work with a variety of commercial products available from IBM and others. Vendors of other programming languages may also offer SOM bindings. Check with your language vendor about possible SOM support.

The SOM run-time library

In addition to the SOM Compiler, SOM includes a **run-time library**. This library provides, among other things, a set of classes, methods, and procedures used to create objects and invoke methods on them. The library allows any programming language to use SOM classes (classes developed using SOM) if that language can:

- Call external procedures,
- Store a pointer to a procedure and subsequently invoke that procedure, and
- Map IDL types onto the programming language's native types.

Thus, the user of a SOM class and the implementor of a SOM class needn't use the same programming language, and neither is required to use an object-oriented language. The independence of client language and implementation language also extends to subclassing: a SOM class can be derived from other SOM classes, and the subclass may or may not be implemented in the same language as the parent class(es). Moreover, SOM's run-time environment allows applications to access information about classes dynamically (at run time).

Frameworks provided in the SOMobjects Toolkit

In addition to SOM itself (the SOM Compiler and the SOM run-time library), the SOMobjects Developer Toolkit also provides a set of *frameworks* (class libraries) that can be used in developing object-oriented applications. These include Distributed SOM, the Interface Repository Framework, the Persistence Framework, the Replication Framework, the Emitter Framework, and the Metaclass Framework described below.

Distributed SOM

Distributed SOM (or **DSOM**) allows application programs to access SOM objects across address spaces. That is, application programs can access objects in other processes, even on different machines. DSOM provides this transparent access to remote objects through its Object Request Broker (ORB): the location and implementation of the object are hidden from the client, and the client accesses the object as if were local. The current release of DSOM supports distribution of objects among processes within a workstation, and across a local area network consisting of AIX systems, OS/2 systems, Windows systems, or some mix of these systems. Future releases may support larger enterprise-wide networks.

Interface Repository Framework

The **Interface Repository** is a database, optionally created and maintained by the SOM Compiler, that holds all the information contained in the IDL description of a class of objects. The **Interface Repository Framework** consists of the 11 classes defined in the CORBA standard for accessing the Interface Repository. Thus, the Interface Repository Framework provides run-time access to all information contained in the IDL description of a class of objects. Type information is available as **TypeCode**s — a CORBA-defined way of encoding the complete description of any data type that can be constructed in IDL.

Persistence Framework

The **Persistence Framework** is a collection of SOM classes that provide methods for saving objects (either in a file or in a more specialized repository) and later restoring them. This means that the state of an object can be preserved beyond the termination of the process that creates it. This facility is useful for constructing object-oriented databases, spreadsheets, and so forth. The Persistence Framework includes the following features:

- Objects can be stored singly or in groups.
- Objects can be stored in default formats or in specially designed formats.
- Objects of arbitrary complexity can be saved and restored.

Replication Framework

The **Replication Framework** is a collection of SOM classes that allows a replica (copy) of an object to exist in multiple address spaces, while maintaining a single-copy image. In other words, an object can be replicated in several different processes, while logically it behaves as a single copy. Updates to any copy are propagated immediately to all other copies. The Replication Framework handles locking, synchronization, and update propagation, and guarantees mutual consistency among the replicas. The Replication Framework includes these important features:

- Good response times for both readers and writers,
- Fault-tolerance against node failures and message loss,
- Simple coding rules (that can be automated) for building replicated objects,
- Graceful degradation under wide-area networks, and
- Minimal overhead when replication is not activated.

Emitter Framework

The **Emitter Framework** is a collection of SOM classes that allows programmers to write their own emitters. *Emitter* is a general term used to describe a back-end output component of the SOM Compiler. Each emitter takes as input information about an interface, generated by the SOM Compiler as it processes an IDL specification, and produces output organized in a different format. SOM provides a set of emitters that generate the binding files for C and C++ programming (header files and implementation templates). In addition, users may wish to write their own special-purpose emitters. For example, an implementor could write an emitter to produce documentation files or binding files for programming languages other than C/C++. The Emitter Framework is separately documented in the *SOMobjects Developer Toolkit: Emitter Framework Guide and Reference.*

Metaclass Framework

Finally, the **Metaclass Framework** is a collection of SOM metaclasses that provide functionality that may be useful to SOM class designers for modifying the default semantics of method invocation and object creation. These metaclasses are described in Chapter 10, “The Metaclass Framework,” of the *SOMobjects Developer Toolkit Users Guide.*

1.3 What's New in SOMobjects Version 2.1

Version 2.1 of the SOMobjects Developer Toolkit provides enhanced capabilities and improved performance for both SOM and DSOM. In addition, the Toolkit now includes support for DirectToSOM (DTS) C++ compilers. New metaclasses in the Metaclass Framework allow class implementors to define classes that automatically possess certain convenience facilities. Also, TCP/IP support is available for Windows users, and OS/2 users can choose 32–bit TCP/IP.

In particular, SOMobjects Version 2.1 offers the following additions over Version 2.0:

General enhancements

- C++ programmers can use DirectToSOM (DTS) C++ compilers (available from independent software vendors) as an alternative to the SOMobjects Toolkit's C++ bindings. (A DTS C++ compiler uses SOM classes to implement C++ objects.) The support provided by SOMobjects for DTS C++ consists of various enhancements to the SOM API (useful to SOM programmers in general), and a new emitter that produces DTS C++ header files corresponding to SOM IDL. DTS C++ programs #include these ".hh" header files, which are emitted as described under "Generating binding files" in Chapter 4, "SOM IDL and the SOM Compiler."
- With this release, TCP/IP support is now available for SOMobjects For Windows. Also, for OS/2 users only, SOMobjects now supports the 32–bit TCP/IP version 2.0, which offers greater performance over the 16–bit TCP/IP version 1.2.1. These are described in your *SOMobjects Installation/Configuration Guide*.

SOMobjects enhancements

- A new default process whereby SOMobjects initializes and destroys objects more efficiently (using the **somDefaultInit** and **somDestruct** methods). See "Initializing and Uninitializing Objects" in Chapter 5, "Implementing Classes in SOM."
- New support for C++-style copy constructors and assignment operators. As with C++, SOM provides reasonable defaults for these, but a class implementor may want to provide specialized implementations. See the **somDefaultConstCopyInit** and **somDefaultConstAssign** methods in the *SOMobjects Developer Toolkit Programmers Reference Manual*.
- A new kind of method, **nonstatic**, that is similar to a C++ nonstatic member function. It is normally invoked using offset resolution but can use any form of SOMobjects method resolution. See "The four kinds of SOM methods" in Chapter 5, "Implementing Classes in SOM" and the **nonstatic** modifier under "Modifier statements" in Chapter 4, "SOM IDL and the SOM Compiler."
- A new kind of data, **staticdata**, that is similar to C++ static data members. A **staticdata** variable is not stored in an object; rather, the ClassData structure of the implementing class contains a pointer to the variable. See the **staticdata** modifier under "Modifier statements" in Chapter 4, "SOM IDL and the SOM Compiler."
- Two new modifiers, **somallocate** and **somdeallocate**, used to indicate that a user-written procedure should be executed to allocate/deallocate memory for class instances when the **somAllocate** or **somDeallocate** method is invoked. See these modifiers under "Modifier statements" in Chapter 4, "SOM IDL and the SOM Compiler."
- The ability to "cast" objects. See the methods **somCastObj** and **somResetObj**.
- New support for loading/unloading class libraries. (This was introduced in SOMobjects For Windows and is now applicable on OS/2 as well.) See the **SOM_ClassLibrary** macro in the *SOMobjects Developer Toolkit Programmers Reference Manual*.

DSOM enhancements

- A new command-line tool, **dsom**, for managing DSOM servers (starting, restarting, stopping, querying, and so forth). See "The **dsom** server manager utility" in Chapter 6, "Distributed SOM." In addition, a corresponding programmatic interface is provided by methods of the new **SOMDServerMgr** class. See that class and its methods in the *SOMobjects Developer Toolkit Programmers Reference Manual*.
- A new command-line tool, **somdchk**, that performs environment validation. See "Verifying the DSOM environment with **somdchk**" in Chapter 6, "Distributed SOM."
- New SOM IDL modifiers for memory management of parameters: **memory_management = corba**, **caller_owns_parameters, caller_owns_result, object_owns_parameters,** and **object_owns_result.** The individual modifiers are described under "Modifier statements" in Chapter 4, "SOM IDL and the SOM Compiler." See "Memory management" in Chapter 6, "Distributed SOM," for a complete discussion of the subject. This memory-management support also includes the new method **somdReleaseResources** and the functions **somdExceptionFree** and **SOMD_NoORBfree**, described in the *SOMobjects Developer Toolkit Programmers Reference Manual*.
- A graphical user interface to the DSOM 'regimpl' utility on OS/2, called **pregimpl**. See "Registration steps using 'pregimpl' or 'wregimpl'," in Chapter 6, "Distributed SOM."
- Support for the CORBA constant OBJECT_NIL. In addition, the **is_nil** method of **SOMDObject** can now be used for local objects as well as NULL pointers.
- Support for passing self-referential structs and unions (those valid in IDL) in remote method calls.
- Support for local/remote transparency in DSOM's object-destruction methods. See "Destroying remote objects."
- Ability for users to define customized base proxy classes. See "Customizing the default base proxy class."
- Ability for users to specify an upper limit on the number of threads that a server can spawn, via the SOMDNUMTHREADS environment variable. See this variable under "Preparing the environment."
- Improved error handling and performance improvements.
- More sample programs distributed with the SOMobjects Toolkit.

Metaclass Framework

- **SOMMBeforeAfter** — a metaclass that enables the programming of "before/after" metaclasses, whose instances execute a particular method before and after each method invocation. See Chapter 10, "The Metaclass Framework," for information about the new Metaclass Framework. Individual metaclasses, along with related classes and methods, are documented in the *SOMobjects Developer Toolkit Programmers Reference Manual*.
- **SOMMTraced** — a utility metaclass to do tracing.
- **SOMRReplicable** — a utility metaclass to ease the use of the Replication Framework.

New restrictions and deprecated methods

While implementing the Metaclass Framework, IBM learned that metaclasses must be programmed so that the capabilities they implement will be preserved when various metaclasses are combined (using multiple inheritance) into a SOM-derived metaclass. To assure this result, the Metaclass Framework metaclasses have been programmed using a "Cooperation Framework." However, IBM is not yet ready to include the Cooperation Framework among the officially supported features of SOMobjects.

To prevent user-defined metaclasses from interfering with the operation of the Cooperation Framework and consequently with the Metaclass Framework, SOMobjects programmers are strongly urged to observe the following restriction when programming new metaclasses:

- User-defined metaclasses can introduce new class methods and class variables, but should *not* override any of the methods introduced by the **SOMClass** class.

SOMobjects users whose metaclass programming requirements cannot be met within the above restrictions will be given access to the Cooperation Framework and its documentation. Note, however, that metaclasses developed using the Cooperation Framework may require reprogramming when an officially supported Cooperation Framework is later introduced.

In addition, use of a number of (public) methods introduced by **SOMClass** is now deprecated because they are useful only from overridden **SOMClass** methods. These methods are listed under the heading “Deprecated methods” in the documentation for **SOMClass** within the *SOMobjects Developer Toolkit Programmers Reference Manual*, until such time as SOMobjects is ready to officially provide a framework within which their use will not interfere with the internal operation of SOMobjects itself.

1.4 Overview of this book

Chapter 2 contains a Tutorial with five examples that illustrate techniques for implementing classes in SOM. *All* readers should cover this tutorial.

Chapter 3 describes how an application program creates instances of a SOM class, how it invokes methods, and so on. For readers interested only in *using* SOM classes, rather than building them, chapters 2 and 3 may provide all of the information they need to begin using SOM classes.

Chapter 4 contains explanations and examples for all valid SOM IDL syntax components. It then presents the SOM Compiler, including all compiler options and related environment variables. All class implementors will need to reference this chapter.

Chapter 5 provides more comprehensive information about the SOM system itself, including operation of the SOM run-time environment, inheritance, and method resolution. This chapter also describes how to create language-neutral class libraries using SOM. In addition, it contains some advanced topics for customizing SOM to better suit the needs of a particular application. All class implementors will need to reference this chapter.

Chapter 6 describes **Distributed SOM** and how to use it to access objects across address spaces, even on different machines. Chapter 6 also describes how to customize DSOM.

Chapter 7 describes the **Interface Repository Framework** of classes supplied with the SOMobjects Toolkit.

Chapter 8 describes the **Persistence Framework** of classes supplied with the SOMobjects Toolkit and how to use it to save objects (either in a file or in a more specialized repository) and later restore them.

Chapter 9 describes the **Replication Framework** of classes supplied with the SOMobjects Toolkit and how to use it to create replicas (copies) of an object in multiple address spaces while maintaining a single-copy image.

Chapter 10 describes the **Metaclass Framework** and some utility metaclasses that SOM provides to assist users in deriving new classes with special abilities to execute “before” and “after” operations when a method call occurs, as well as other capabilities for modifying the default semantics of method invocation and object creation.

Chapter 11 discusses a large group of “collection classes” that are provided with the SOM system. These collection classes and related methods implement most of the common data structures that programmers need.

Chapter 12 describes the **Event Management Framework**, which allows grouping of all application events and waiting on multiple events in one place. The Event Manager is used by both DSOM and the Replication Framework.

Appendix A describes service and technical support policies for the SOMobjects Toolkit, and contains lists of the error codes and messages that can be issued by the SOM kernel or by the various frameworks.

Appendix B gives instructions for automatically converting class definition files from OIDL syntax (the interface definition language used by the previous release of SOM) to IDL syntax (the language prescribed by the CORBA standard).

Appendix C contains the SOM IDL language grammar.

Appendix D provides extensive information and examples for subclassing the Persistence Framework.

Appendix E describes how to subclass a “Sockets” class that facilitates inter-process communications required by the DSOM, Replication, and Event Management Frameworks.

Appendix F presents the ‘emitcom’ program, which creates bindings that allow SOM classes to be used with COM (Microsoft’s component interface model).

Chapter 2. Tutorial for Implementing SOM Classes

Contents

2.1 Basic Concepts of the System Object Model (SOM) . . . . . 2 – 1
Development of the Tutorial examples . . . . . 2 – 5
2.2 Basic Steps for Implementing SOM Classes . . . . . 2 – 6
Example 1 — Implementing a Simple Class with One Method . . . . . 2 – 7
Example 2 — Adding an Attribute to the Hello class . . . . . 2 – 13
Attributes vs instance variables . . . . . 2 – 15
Example 3 — Overriding an Inherited Method . . . . . 2 – 17
Example 4 — Initializing a SOM Object . . . . . 2 – 20
Example 5 — Using Multiple Inheritance . . . . . 2 – 22

Chapter 2. Tutorial for Implementing SOM Classes

This tutorial contains five examples showing how SOM classes can be implemented to achieve various functionality. Obviously, for any person who wishes to become a class implementor, this tutorial is essential. However, even for those programmers who expect only to *use* SOM classes that were implemented by others, the tutorial is also necessary, as it presents several concepts that will help clarify the process of using SOM classes.

2.1 Basic Concepts of the System Object Model (SOM)

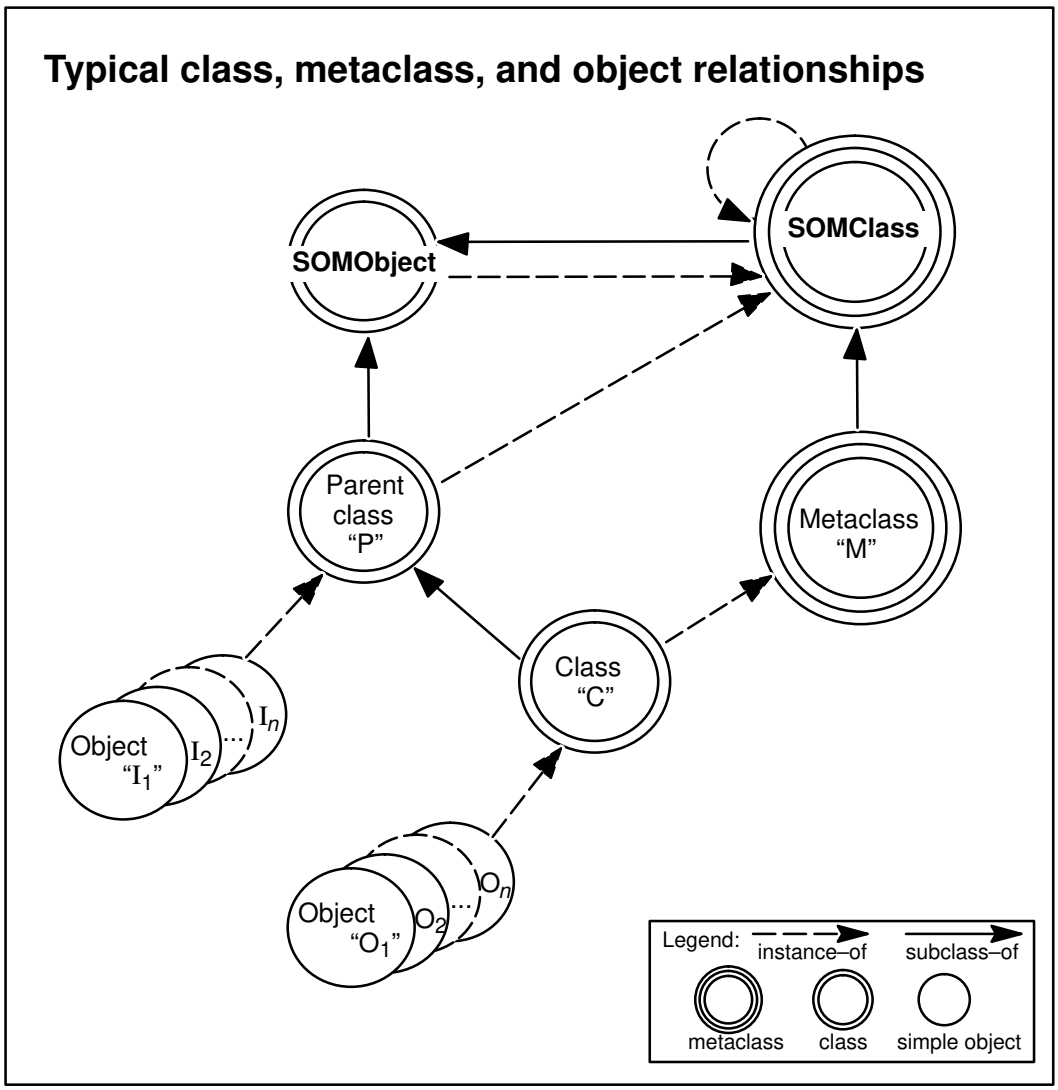
The **System Object Model (SOM),** provided by the **SOMobjects Developer Toolkit**, is a set of libraries, utilities, and conventions used to create binary class libraries that can be used by application programs written in various object-oriented programming languages, such as C++ and Smalltalk, or in traditional procedural languages, such as C and Cobol. The following paragraphs introduce some of the basic terminology used when creating classes in SOM:

- An *object* is an OOP entity that has *behavior* (its *methods* or operations) and *state* (its data values). In SOM, an object is a run-time entity with a specific set of methods and instance variables. The methods are used by a client programmer to make the object exhibit behavior (that is, to do something), and the instance variables are used by the object to store its state. (The state of an object can change over time, which allows the object's behavior to change.) When a method is invoked on an object, the object is said to be the *receiver* or *target* of the method call.
- An object's *implementation* is determined by the procedures that execute its methods, and by the type and layout of its instance variables. The procedures and instance variables that implement an object are normally *encapsulated* (hidden from the caller), so a program can use the object's methods without knowing anything about how those methods are implemented. Instead, a user is given access to the object's methods through its *interface* (a description of the methods in terms of the data elements required as input and the type of value each method returns).
- An interface through which an object may be manipulated is represented by an *object type*. That is, by declaring a type for an object variable, a programmer specifies an interface that can be used to access that object. *SOM IDL* (the **SOM Interface Definition Language**) is used to define object interfaces. The *interface names* used in these IDL definitions are also the type names used by programmers when typing SOM object variables.
- In SOM, as in most approaches to object-oriented programming, a *class* defines the implementation of objects. That is, the implementation of any SOM object (as well as its interface) is defined by some specific SOM class. A class definition begins with an IDL specification of the interface to its objects, and the name of this interface is used as the class name as well. Each object of a given class may also be called an *instance* of the class, or an *instantiation* of the class.
- *Inheritance,* or *class derivation*, is a technique for developing new classes from existing classes. The original class is called the *base* class, or the *parent* class, or sometimes the direct *ancestor* class. The derived class is called a *child* class or a *subclass*. The primary advantage of inheritance is that a derived class inherits all of its parent's methods and instance variables. Also through inheritance, a new class can *override* (or redefine) methods of its parent, in order to provide enhanced functionality as needed. In addition, a derived class can introduce new methods of its own. If a class results from several generations of successive class derivation, that class "knows" all of its ancestors's methods (whether overridden or not), and an object (or instance) of that class can execute any of those methods.

- SOM classes can also take advantage of *multiple inheritance,* which means that a new class is jointly derived from two or more parent classes. In this case, the derived class inherits methods from all of its parents (and all of its ancestors), giving it greatly expanded capabilities. In the event that different parents have methods of the same name that execute differently, SOM provides ways for avoiding conflicts.
- In the SOM run time, classes are themselves objects. That is, classes have their own methods and interfaces, and are themselves defined by other classes. For this reason, a class is often called a *class object*. Likewise, the terms *class methods* and *class variables* are used to distinguish between the methods/variables of a class object vs. those of its instances. (Note that the type of an object is *not* the same as the type of its class, which as a "class object" has its own type.)
- A class that defines the implementation of class objects is called a *metaclass*. Just as an instance of a class is an object, so an instance of a metaclass is a class object. Moreover, just as an ordinary class defines methods that its objects respond to, so a metaclass defines methods that a class object responds to. For example, such methods might involve operations that execute when a class (that is, a class object) is creating an instance of itself (an object). Just as classes are derived from parent classes, so metaclasses can be derived from parent metaclasses, in order to define new functionality for class objects.
- The SOM system contains three primitive classes that are the basis for all subsequent classes:

  SOMObject — the root ancestor class for all SOM classes,
  SOMClass — the root ancestor class for all SOM metaclasses, and
  SOMClassMgr — the class of the SOMClassMgrObject, an object created automatically during SOM initialization, to maintain a registry of existing classes and to assist in dynamic class loading/unloading.

  SOMClass is defined as a subclass (or child) of **SOMObject** and inherits all generic object methods; this is why instances of a metaclass are class *objects* (rather than simply classes) in the SOM run time. The adjacent figure illustrates typical relationships of classes, metaclasses, and objects in the SOM run time. (This illustration does not include the SOMClassMgrObject.)



SOM classes are designed to be *language neutral*. That is, SOM classes can be implemented in one programming language and used in programs of another language. To achieve language neutrality, the *interface* for a class of objects must be defined separately from its *implementation*. That is, defining interface and implementation requires two completely separate steps (plus an intervening compile), as follows:

- An *interface* is the information that a program must know in order to use an object of a particular class. This interface is described in an interface definition (which is also the class definition), using a formal language whose syntax is independent of the programming language used to implement the class's methods. For SOM classes, this is the SOM Interface Definition Language (SOM IDL). The interface is defined in a file known as the *IDL source file* (or, using its extension, this is often called the *.idl file*).

  An interface definition is specified within the *interface declaration* (or *interface statement*) of the .idl file, which includes:

  (a) the interface name (or class name) and the name(s) of the class's parent(s), and
  (b) the names of the class's attributes and the signatures of its new methods. (Recall that the complete set of available methods also includes all inherited methods.)

  Each *method signature* includes the method name, and the type and order of its arguments, as well as the type of its return value (if any). *Attributes* are instance variables for

which “set” and “get” methods will automatically be defined, for use by the application program. (By contrast, instance variables that are not attributes are hidden from the user.)

- Once the IDL source file is complete, the *SOM Compiler* is used to analyze the .idl file and create the *implementation template file*, within which the class implementation will be defined. Before issuing the SOM Compiler command, **sc** (or **somc** on Windows), the class implementor can set an environment variable that determines which *emitters* (output-generating programs) the SOM Compiler will call and, consequently, which programming language and operating system the resulting *binding files* will relate to. (Alternatively, this emitter information can be placed on the command line for **sc** or **somc.**)

  In addition to the implementation template file itself, the binding files include two language-specific header files that will be #included in the implementation template file and in application program files. The header files define many useful SOM macros, functions, and procedures that can be invoked from the files that include the header files.

- The *implementation* of a class is done by the class implementor in the *implementation template file* (often called just the *implementation file* or the *template file*). As produced by the SOM Compiler, the template file contains *stub procedures* for each method of the class. These are incomplete method procedures that the class implementor uses as a basis for implementing the class by writing the corresponding code in the programming language of choice.

In summary, the process of *implementing a SOM class* includes using the SOM IDL syntax to create an IDL source file that specifies the interface to a class of objects — that is, the methods and attributes that a program can use to manipulate an object of that class. The SOM Compiler is then run to produce an implementation template file and two binding (header) files that are specific to the designated programming language and operating system. Finally, the class implementor writes language-specific code in the template file to implement the method procedures.

At this point, the next step is to write the application (or client) program(s) that use the objects and methods of the newly implemented class. (Observe, here, that a programmer could write an application program using a class implemented entirely by someone else.) If not done previously, the SOM compiler is run to generate usage bindings for the new class, as appropriate for the language used by the client program (which may be different from the language in which the class was implemented). After the client program is finished, the programmer compiles and links it using a language-specific compiler, and then executes the program. (Notice again, the client program can invoke methods on objects of the SOM class without knowing how those methods are implemented.)

Development of the Tutorial examples

- **Example 1 — Implementing a simple class with one method**
  Prints a default message when the “sayHello” method is invoked on an object of the “Hello” class.
- **Example 2 — Adding an attribute to the Hello class**
  Defines a “msg” attribute for the “sayHello” method to use. The client program “sets” a message; then the “sayHello” method “gets” the message and prints it. (There is no defined message when an object of the “Hello” class is first created.)
- **Example 3 — Overriding an inherited method**
  Overrides the SOMobjects method **somPrintSelf** so that invoking this method on an object of the “Hello” class will not only display the class name and the object’s location, but will also include the object’s message attribute.
- **Example 4 — Initializing a SOM object**
  Overrides the default initialization method, **somDefaultInit**, to illustrate how an object’s instance variables can be initialized when the object is created.
- **Example 5 — Using multiple inheritance**
  Extends the “Hello” class to provide it with multiple inheritance (from the “Disk” and “Printer” classes). The “Hello” interface defines an enum and an “output” attribute that takes its value from the enum (either “screen”, “printer”, or “disk”). The client program “sets” the form of “output” before invoking the “sayHello” method to send a “msg” (defined as in Example 4).

2.2 Basic Steps for Implementing SOM Classes

Implementing and using SOM classes in C or C++ involves the following steps, which are explicitly illustrated in the examples of this tutorial:

1. Define the interface to objects of the new class (that is, the interface declaration), by creating a **.idl** file.
2. Run the SOM Compiler on the .idl file (by issuing the **sc** command on AIX or OS/2, or by issuing the **somc** command on Windows) to produce the following binding files:
   - Template implementation file
     a **.c** file for C programmers, or
     a **.C** file (on AIX) or a **.cpp** file (on OS/2 or Windows) for C++ programmers;
   - Header file to be included in the implementation file
     a **.ih** file for C programmers, or
     a **.xih** file for C++ programmers; and
   - Header file to be included in client programs that use the class
     a **.h** file for C clients, or
     a **.xh** file for C++ clients.

   To specify whether the SOM Compiler should produce C or C++ bindings, set the value of the SMEMIT environment variable or use the "**–s**" option of the **sc** or **somc** command, as described in Section 4.3, "The SOM Compiler." By default, the SOM Compiler produces C bindings.
3. Customize the implementation, by adding code to the template implementation file.
4. Create a client program that uses the class.
5. Compile and link the client code with the class implementation, using a C or C++ compiler.
6. Execute the client program.

The following examples illustrate appropriate syntax for defining interface declarations in a .idl file, including designating the methods that the class's instances will perform. In addition, the example template implementation files contain typical code that the SOM Compiler produces. Explanations accompanying each example discuss topics that are significant to the particular example; full explanations of the SOM IDL syntax are contained in Chapter 4, "SOM IDL and the SOM Compiler." Customization of each implementation file (step 3) is illustrated in both C and C++.

Notes: (1) The Tutorial assumes you will work through the examples in order. If you do *not* do so, the code that the SOM Compiler generates from your revised .idl file may vary slightly from what you see in the Tutorial.
(2) When the SOMobjects Toolkit is installed, a choice is made between "somcorba" and "somstars" for the style of C bindings the SOM Compiler will generate. The Tutorial examples use the "somcorba" style, where an interface name used as a type indicates a *pointer* to an object, as required by strict CORBA bindings. Consequently, as the examples show, a "*****" does not explicitly appear for types that are pointers to objects. If your system was installed for "somstars" C bindings, you can set the environment variable SMADDSTAR=1 or use the SOM Compiler option "–maddstar" to request bindings that use explicit pointer stars. For more information, see "Declaring object variables" in Chapter 3,"Using SOM Classes in Client Programs" and "Object types" in Chapter 4, "SOM IDL and the SOM Compiler."

Example 1 — Implementing a Simple Class with One Method

Example 1 defines a class "Hello" which introduces one new method, "sayHello". When invoked from a client program, the "sayHello" method will print the fixed string "Hello, World!" The example follows the six steps described in the preceding topic, "Basic Steps for Implementing SOM Classes."

1) Define the interface to class "Hello", which inherits methods from the root class **SOMObject** and introduces one new method, "sayHello". Define these IDL specifications in the file "hello.idl".

   The "interface" statement introduces the name of a new class and any parents (base classes) it may have (here, the root class **SOMObject**). The body of the interface declaration introduces the method "sayHello." Observe that method declarations in IDL have syntax similar to C and C++ function prototypes:

```
#include <somobj.idl>  //# Get the parent class definition.

interface Hello : SOMObject
/*  This is a simple class that demonstrates how to define the
 *  interface to a new class of objects in SOM IDL.
 */
{
    void sayHello();
    // This method outputs the string "Hello, World!".
       /* On Windows, use:   string sayHello();
        * This method returns the string "Hello, World!". */
};
```

   Note that the method "sayHello" has no (explicit) arguments and returns no value (except on Windows, which returns a string). The characters "//" start a line comment, which finishes at the end of the line. The characters "/*" start a block comment that finishes with "*/". Block comments do not nest. The two comment styles can be used interchangeably. Throw-away comments are also permitted in a .idl file; they are ignored by the SOM Compiler. Throw-away comments start with the characters "//#" and terminate at the end of the line.

   Note: For simplicity, this IDL fragment does not include a **releaseorder** modifier; consequently, the SOM Compiler will issue a warning for the method "sayHello". For directions on using the **releaseorder** modifier to remove this warning, see the topic "Modifier statements" in Chapter 4, "SOM IDL and the SOM Compiler." (The warning does not prohibit continued use of the .idl file.)

2) Run the SOM Compiler to produce binding files and an implementation template. That is, issue the **sc** command on AIX or OS/2, as follows:

```
> sc -s"c;h;ih" hello.idl      (for C bindings on AIX or OS/2)

> sc -s"xc;xh;xih" hello.idl   (for C++ bindings on AIX or OS/2)
```

   On Windows, issue the **somc** command (started from the Run option on the File menu):

```
> somc -sc;h;ih hello.idl      (for C bindings on Windows)

> somc -sxc;xh;xih hello.idl   (for C++ bindings on Windows)
```

   When set to generate C binding files, the SOM Compiler generates the following template implementation file, named "hello.c". The template implementation file contains *stub procedures* for each new method; these are procedures whose bodies are largely vacuous, to be filled in by the implementor. (Unimportant details have been removed for this tutorial.)

Under AIX or **OS/2**:

```
#include <hello.ih>

/*
 *  This method outputs the string "Hello, World!".
 */


SOM_Scope void   SOMLINK sayHello(Hello somSelf, Environment *ev)
{
    /* HelloData *somThis = HelloGetData(somSelf); */
    HelloMethodDebug("Hello", "sayHello");
}
```

Under Windows:

```
#include <hello.ih>

/*
 *  This method returns the string "Hello, World!".
 */


SOM_Scope string  SOMLINK sayHello(Hello somSelf, Environment *ev)
{
    /* HelloData *somThis = HelloGetData(somSelf); */
    HelloMethodDebug("Hello", "sayHello");
}
```

The terms "SOM_Scope" and "SOMLINK" appear in the prototype for all stub procedures, but they are defined by SOM and are not of interest to the developer. In the method procedure for the "sayHello" method, "somSelf" is a pointer to the target object (here, an instance of the class "Hello") that will respond to the method. A "somSelf" parameter appears in the procedure prototype for every method, since SOM requires every method to act on some object.

The *target object* is always the first parameter of a method's procedure, although it should *not* be included in the method's IDL specification. The second parameter (which also is not included in the method's IDL specification) is the parameter **(Environment *ev)**. This parameter can be used by the method to return exception information if the method encounters an error. (Contrast the prototype for the "sayHello" method in steps 1 and 2 above.)

The remaining lines of the template above are not pertinent at this point. (For those interested, they are discussed in section 5.4 of Chapter 5, "Implementing Classes in SOM.") The file is now ready for customization with the C code needed to implement method "sayHello".

When set to generate C++ binding files, the SOM Compiler generates an implementation template file, "hello.C" (on AIX) or "hello.cpp (on OS/2 or Windows), similar to the one above. (Chapter 5 discusses the implementation template in more detail.)

Recall that, in addition to generating a template implementation file as shown above, the SOM Compiler also generates implementation bindings (in a header file to be included in the implementation file) and usage bindings (in a header file to be included in client programs). These files are named "hello.ih" and "hello.h" for C bindings, and are "hello.xih" and "hello.xh" for C++ bindings. Notice that the "hello.c" file shown above includes the "hello.ih" implementation binding file.

3) Customize the implementation, by adding code to the template implementation file.

   Modify the body of the "sayHello" method procedure in the "hello.c" (or, for C++,"hello.C" on AIX, "hello.cpp" on OS/2) implementation file so that the "sayHello" method prints "Hello, World!":

```
SOM_Scope void   SOMLINK sayHello(Hello somSelf, Environment *ev)
{
    /* HelloData *somThis = HelloGetData(somSelf); */
    HelloMethodDebug("Hello","sayHello");

    printf("Hello, World!\n");
        /*  On Windows, use:  return ("Hello, World!");   */
}
```

4) Create a client program that uses the class.

   Write a program "main" that creates an instance (object) of the "Hello" class and invokes the method "sayHello" on that object.

   Under AIX or OS/2:
   A C programmer would write the following program in "main.c", which uses the bindings defined in the "hello.h" header file:

```
#include <hello.h>

int main(int argc, char *argv[])
{
    /* Declare a variable to point to an instance of Hello */
    Hello obj;

    /* Create an instance of the Hello class */
    obj = HelloNew();

    /* Execute the "sayHello" method */
    _sayHello(obj, somGetGlobalEnvironment());

    /* Free the instance: */
    _somFree(obj);
    return (0);
}
```

   Notice the statement `obj = HelloNew();` The "hello.h" header file automatically contains the SOM-defined macro <*className*>**New()**, which is used to create an instance of the <*className*> class (here, the "Hello" class).

   Also notice that, in C, a method is invoked on an object by using the form:

   _<*methodName*>**(**<*objectName*>, <*environment_arg*>, <*other_method_args*>**)**

   as used above in the statement `_sayHello(obj, somGetGlobalEnvironment())`. As shown in this example, the SOM-provided **somGetGlobalEnvironment** function can be used to supply the (**Environment** *) argument of the method.

   Finally, the code uses the method **somFree**, which SOM also provides, to free the object created by `HelloNew()`. Notice that **somFree** does not require an (**Environment** *) argument. This is because the method procedures for some of the classes in the SOMobjects Toolkit (including **SOMObject**, **SOMClass**, and **SOMClassMgr**) do not have an Environment parameter, to ensure compatibility with the first release of SOM. The documentation for each SOM-kernel method in the *SOMobjects Developer Toolkit Programmers Reference Manual* indicates whether an Environment parameter is used.

A C++ programmer would write the following program in “main.C” (on AIX) or “main.cpp” (on OS/2), using the bindings defined in the “hello.xh” header file:

```
#include <hello.xh>

int main(int argc, char *argv[])
{
    /* Declare a variable to point to an instance of Hello */
    Hello *obj;

    /* Create an instance of the Hello class */
    obj = new Hello;

    /* Execute the "sayHello" method */
    obj->sayHello(somGetGlobalEnvironment());

    obj->somFree();
    return (0);
}
```

Notice that the only argument passed to the “sayHello” method by a C++ client program is the **Environment** pointer. (Contrast this with the invocation of “sayHello” in the C client program, above.)

Under Windows:
The subsequent “main.c” file fragment uses bindings defined in the “hello.h” and “windows.h” header files, and contains C code illustrating the WinMain and WndProc procedures that are pertinent to Step 4. The macro **SOM_MainProgram** is recommended as the first line of executed code in each executable Windows program, such as WinMain (note: it does not work from a DLL). **SOM_MainProgram** combines execution of the **somMainProgram** function with scheduling of the **somEnvironmentEnd** function, which frees classes introduced by your application when they are no longer needed.

```
#include <windows.h>
#include <hello.h>

int PASCAL WinMain (HANDLE hInstance, HANDLE hPrevInstance,
                    LPSTR lpszCmdLine, int nCmdShow)
     {
     < variable declarations here >

     SOM_MainProgram();
     /*  Registers an exit procedure to ensure cleanup upon
      *  normal program termination. */

     < rest of WinMain here >
     }
```

Notice that the following WndProc code handles three messages: WM_CREATE, WM_PAINT, and WM_DESTROY. It lets the Windows default procedure handle the rest. The “Hello” object is created in response to the Window creation message (WM_CREATE) and is freed in response to the Window destroy message (WM_DESTROY). The “sayHello” method’s greeting is displayed each time the window receives a WM_PAINT message. Notice that “obj” is declared static so that it persists between calls.

```
long FAR PASCAL _export WndProc (HWND hwnd, UINT message,
                                 UINT wParam, LONG lParam)
     {
     HDC         hdc ;
     PAINTSTRUCT ps ;
     RECT        rect ;
     char sBuf [80];
     static Hello obj;

     switch (message)
          {
          case WM_CREATE:
                obj = HelloNew();
                return 0;

          case WM_PAINT:
               hdc = BeginPaint (hwnd, &ps) ;

               GetClientRect (hwnd, &rect) ;

               strcpy(sBuf, _sayHello(obj,
                                   somGetGlobalEnvironment()));
               DrawText (hdc, sBuf, -1, &rect,
                         DT_SINGLELINE | DT_CENTER | DT_VCENTER);

               EndPaint (hwnd, &ps) ;
               return 0 ;

          case WM_DESTROY:
               _somFree(obj);
               PostQuitMessage (0) ;
               return 0 ;
          }
     return DefWindowProc (hwnd, message, wParam, lParam) ;
     }
```

A C++ programmer on Windows would write a “main.cpp” file that uses the bindings defined in the “hello.xh” and “windows.xh” header files, with corresponding WinMain and WndProc code.

5) Compile and link the client code with the class implementation.

   Note: On AIX or OS/2, the environment variable SOMBASE represents the directory in which SOM has been installed.

   Under AIX, for C programmers:

```
> xlc -I. -I$SOMBASE/include -o hello main.c hello.c \
      -L$SOMBASE/lib -lsomtk
```

   Under AIX, for C++ programmers:

```
> xlC -I. -I$SOMBASE/include -o hello main.C hello.C \
      -L$SOMBASE/lib -lsomtk
```

   Under OS/2, for C programmers:

```
> set LIB=%SOMBASE%\lib;%LIB%
> icc -I. -I%SOMBASE%\include -Fe hello main.c hello.c \
      somtk.lib
```

   Under OS/2, for C++ programmers:

```
> set LIB=%SOMBASE%\lib;%LIB%
> icc -I. -I\%SOMBASE%\include -Fe hello main.cpp hello.cpp \
      somtk.lib
```

Under Windows, for C programmers**:**

```
> cl -AL -Zp -D_WIN16 -F 5 -I. -I%SOMBASE%\include \
     -Fehello main.c hello.c llibcew.lib libw.lib somtk.lib
```

where %SOMBASE% is replaced by the directory in which SOM was installed (the default is c:\som). In makefiles, the expression $(SOMBASE) can be used. Note that the –F option to set the stack size is unnecessary if STACKSIZE is specified in a .def file.

Important: In this manual, compile and link commands are demonstrated for Windows using the Visual C++ compiler. Sample Makefiles shipped with SOMobjects For Windows also show how to build with other compilers.

Under Windows, for C++ programmers:

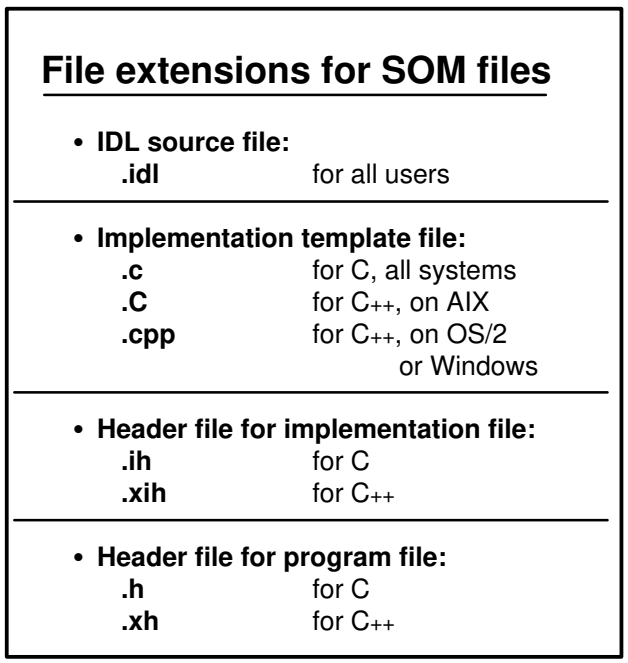
```
> cl -ALu -Zp -D_WIN16 -F 5 -I. -I%SOMBASE%\include \
     -Fehello main.cpp hello.cpp llibcew.lib libw.lib somtk.lib
```

where %SOMBASE% is replaced by the directory in which SOM was installed (the default is c:\som). In makefiles, the expression $(SOMBASE) can be used. Note that the –F option to set the stack size is unnecessary if STACKSIZE is specified in a .def file.

6) Execute the client program.

```
> hello
Hello, World!
```

Example 2 will extend the “Hello” class to introduce an “attribute”.



Example 2 — Adding an Attribute to the Hello class

Example 1 introduced a class “Hello” which has a method “sayHello” that prints the fixed string “Hello, World!” Example 2 extends the “Hello” class so that clients can customize the output from the method “sayHello”.

1) Modify the interface declaration for the class definition in “hello.idl.”

   Class “Hello” is extended by adding an attribute that we call “msg”. Declaring an *attribute* is equivalent to defining “get” and “set” methods. For example, specifying:

```
attribute string msg;
```

   is equivalent to defining the two methods:

```
string _get_msg();
void  _set_msg(in string msg);
```

   Thus, for convenience, an attribute can be used (rather than an instance variable) in order to use the automatically defined “get” and “set” methods without having to write their method procedures. The new interface specification for “Hello” that results from adding attribute “msg” to the “Hello” class is as follows (with some comment lines omitted):

```
#include <somobj.idl>

interface Hello : SOMObject
{
  void  sayHello();

  attribute string msg;
   //# This is equivalent to defining the methods:
   //#     string _get_msg();
   //#      void  _set_msg(string msg);
};
```

2) Re-run the SOM Compiler on the updated .idl file, as in example 1. This produces new header files and updates the existing implementation file, if needed, to reflect changes made to the .idl file. In this example, the implementation file is not modified by the SOM Compiler.

3) Customize the implementation.

   Customize the implementation file by modifying the print statement in the “sayHello” method procedure. This example prints the contents of the “msg” attribute (which must be initialized in the client program) by invoking the “_get_msg” method. Notice that, because the “_get_msg” method name begins with an underscore, the method is invoked with *two* leading underscores (for C only).

```
SOM_Scope void   SOMLINK sayHello(Hello somSelf, Environment *ev)
{
    /* HelloData *somThis = HelloGetData(somSelf); */
    HelloMethodDebug("Hello", "sayHello");

    printf("%s\n", __get_msg(somSelf, ev));

    /* for C++, use  somSelf->_get_msg(ev);  */
}
```

   This implementation assumes that “_set_msg” has been invoked to initialize the “msg” attribute before the “_get_msg” method is invoked by the “sayHello” method. This initialization can be done within the client program.

4) Update the client program.

   Modify the client program so that the "_set_msg" method is invoked to initialize the "msg" attribute before the "sayHello" method is invoked. Notice that, because the "_set_msg" method name begins with an underscore, the C client program invokes the method with *two* leading underscores.

   For C programmers:

```
#include <hello.h>

int main(int argc, char *argv[])
{
    Hello obj;

    obj = HelloNew();

    /* Set the msg text */
    __set_msg(obj, somGetGlobalEnvironment(), "Hello World Again");

    /* Execute the "sayHello" method */
    _sayHello(obj, somGetGlobalEnvironment());

    _somFree(obj);
    return (0);
}
```

   For C++ programmers:

```
#include <hello.xh>

int main(int argc, char *argv[])
{
    Hello *obj;
    obj = new Hello;

    /* Set the msg text */
    obj->_set_msg(somGetGlobalEnvironment(), "Hello World Again");

    /* Execute the "sayHello" method */
    obj->sayHello(somGetGlobalEnvironment());

    obj->somFree();
    return (0);
}
```

5) Compile and link the client program, as before.

6) Execute the client program:

```
> hello
Hello World Again
```

   The next example extends the "Hello" class to override (redefine) one of the methods it inherits from its parent class, **SOMObject**.

Attributes vs instance variables

As an alternative to defining “msg” as an attribute, an instance variable “message” could be introduced, with “set_msg” and “get_msg” methods defined for setting and retrieving its value. Instance variables are declared in an **implementation** statement, as shown below:

```
interface Hello
{
  string get_msg() ;
  void set_msg(in string msg);

#ifdef __SOMIDL__
implementation
{
    string message;
};
#endif
};
```

As demonstrated in this example, one disadvantage to using an instance variable is that the “get_msg” and “set_msg” methods must be defined in the implementation file by the class implementor. For attributes, by contrast, default implementations of the “get” and “set” methods are generated automatically by the SOM Compiler in the .ih and .xih header files.

Note: For some attributes (particularly those involving structures, strings, and pointers) the default implementation generated by the SOM Compiler for the “set” method may not be suitable. This happens because the SOM Compiler only performs a “shallow copy,” which typically is not useful for distributed objects with these types of attributes. In such cases, it is possible to write your own implementations, as you do for any other method, by specifying the “noset/noget” modifiers for the attribute. (See the subtopic “Modifier statements” in Chapter 4, “SOM IDL and the SOM Compiler.”)

Regardless of whether you let the SOM Compiler generate your implementations or not, if access to instance data is required, either from a subclass or a client program, then this access should be facilitated by using an attribute. Otherwise, instance data can be defined in the “implementation” statement as above (using the same syntax as used to declare variables in C or C++), with appropriate methods defined to access it. For more information about “implementation” statements, see the topic “Implementation statements” in Chapter 4.

As an example where instance variables would be used (rather than attributes), consider a class “Date” that provides a method for returning the current date. Suppose the date is represented by three instance variables — “mm”, “dd”, and “yy”. Rather than making “mm”, “dd”, and “yy” attributes (and allowing clients to access them directly), “Date” defines “mm”, “dd”, and “yy” as instance variables in the “implementation” statement, and defines a method “get_date” that converts “mm”, “dd”, and “yy” into a string of the form “mm/dd/yy”:

```
interface Date
{
    string get_date() ;

#ifdef __SOMIDL__
implementation
{
    long mm,dd,yy;
};
#endif
};
```

To access instance variables that a class introduces from within the class implementation file, two forms of notation are available:

somThis–>variableName

or

_variableName

For example, the implementation for “get_date” would likely

access the “mm” instance variable as `somThis->mm` or `_mm`,
access “dd” as `somThis->dd` or `_dd`, and
access “yy” as `somThis->yy` or `_yy`.

In C++ programs, the *_variableName* form is available only if the programmer first defines the macro VARIABLE_MACROS (that is, enter `#define VARIABLE_MACROS`) in the implementation file prior to including the .xih file for the class.

Example 3 — Overriding an Inherited Method

An important aspect of OOP programming is the ability of a subclass to replace an inherited method implementation with a new implementation especially appropriate to its instances. This is called *overriding* a method. Sometimes, a class may introduce methods that every descendant class is expected to override. For example, **SOMObject** introduces the **somPrintSelf** method, and a good SOM programmer will generally override this method when implementing a new class.

The purpose of **somPrintSelf** is to print a brief description of an object. The method can be useful when debugging an application that deals with a number of objects of the same class — assuming the class designer has overridden **somPrintSelf** with a message that is useful in distinguishing different objects of the class. For example, the implementation of **somPrintSelf** provided by **SOMObject** simply prints the class of the object and its address in memory. **SOMClass** overrides this method so that, when **somPrintSelf** is invoked on a class object, the name of the class will print.

This example illustrates how **somPrintSelf** might be overridden for the "Hello" class. An important identifying characteristic of "Hello" objects is the message they hold; thus, the following steps illustrate how **somPrintSelf** could be overridden in "Hello" to provide this information.

1) Modify the interface declaration in "hello.idl."

To override the **somPrintSelf** method in "Hello", additional information must be provided in "hello.idl" in the form of an **implementation** statement, which gives extra information about the class, its methods and attributes, and any instance variables. (The previous examples omitted the optional "implementation" statement, because it was not needed.)

In the current example, the "implementation" statement introduces the *modifiers* for the "Hello" class, as follows.

```
#include <somobj.idl>

interface Hello : SOMObject
{
  void sayHello();

  attribute string msg;

#ifdef __SOMIDL__
implementation
{
    //# Method Modifiers:
    somPrintSelf: override;
    // Override the inherited implementation of somPrintSelf.
};
#endif

};
```

Here, "somPrintSelf:" introduces a list of *modifiers* for the (inherited) **somPrintSelf** method in the class "Hello". Modifiers are like C/C++ #pragma commands and give specific implementation details to the compiler. This example uses only one modifier, "override". Because of the "override" modifier, when **somPrintSelf** is invoked on an instance of class "Hello", Hello's implementation of **somPrintSelf** (defined in the implementation file) will be called, instead of the implementation inherited from the parent class, **SOMObject**.

The “#ifdef _ _SOMIDL_ _” and “#endif” are standard C and C++ preprocessor commands that cause the “implementation” statement to be read only when using the SOM IDL compiler (and not some other IDL compiler).

2) Re-run the SOM Compiler on the updated .idl file, as before. The SOM Compiler extends the existing implementation file from Example 2 to include new stub procedures as needed (in this case, for **somPrintSelf**). Below is a shortened version of the C language implementation file as updated by the SOM Compiler; C++ implementation files are similarly revised. Notice that the code previously added to the “sayHello” method is not disturbed when the SOM Compiler updates the implementation file.

```
#include <hello.ih>

SOM_Scope void   SOMLINK sayHello(Hello somSelf, Environment *ev)
{
    /* HelloData *somThis = HelloGetData(somSelf); */
    HelloMethodDebug("Hello","sayHello");

    printf("%s\n", __get_msg(somSelf, ev));
}

SOM_Scope void   SOMLINK somPrintSelf(Hello somSelf)
{
    HelloData *somThis = HelloGetData(somSelf);
    HelloMethodDebug("hello","somPrintSelf");

    Hello_parent_SOMObject_somPrintSelf(somSelf);
}
```

Note that the SOM Compiler added code allowing the “Hello” class to redefine **somPrintSelf**. The SOM Compiler provides a default implementation for overriding the **somPrintSelf** method. This default implementation simply calls the “parent method” [the procedure that the parent class of “Hello” (**SOMObject**) uses to implement the **somPrintSelf** method]. This parent method call is accomplished by the macro **`Hello_parent_SOMObject_somPrintSelf`**, defined in “hello.ih.”

Notice that the stub procedure for overriding the **somPrintSelf** method does not include an Environment parameter. This is because **somPrintSelf** is introduced by **SOMObject**, which does not include the Environment parameter in any of its methods (to ensure backward compatibility). The signature for a method cannot change after it has been introduced.

3) Customize the implementation.

Within the new **somPrintSelf** method procedure, display a brief description of the object, appropriate to “Hello” objects. Note that the parent method call is not needed, so it has been deleted. Also, direct access to instance data introduced by the “Hello” class is not required, so the assignment to “somThis” has been commented out (see the first line of the procedure).

```
SOM_Scope void   SOMLINK somPrintSelf(Hello somSelf)
{
    /* HelloData *somThis = HelloGetData(somSelf); */
    HelloMethodDebug("Hello","somPrintSelf");

    somPrintf("-- a %s object at location %X with msg: %s\n",
              _somGetClassName(somSelf),
              somSelf,
              __get_msg(somSelf,0));
}
```

4) Update the client program to illustrate the change (also notice the new message text):

   For C programmers:

```
#include <hello.h>

int main(int argc, char *argv[])
{
    Hello obj;
    Environment *ev = somGetGlobalEnvironment();

    obj = HelloNew();

    /* Set the msg text */
    __set_msg(obj, ev, "Hi There");

    /* Execute the "somPrintSelf" method */
    _somPrintSelf(obj);

    _somFree(obj);
    return (0);
}
```

   For C++ programmers:

```
#include <hello.xh>

int main(int argc, char *argv[])
{
    Hello *obj;
    Environment *ev = somGetGlobalEnvironment();

    obj = new Hello;

    /* Set the msg text */
    __set_msg(obj, ev, "Hi There");

    /* Execute the "somPrintSelf" method */
     obj->somPrintSelf();

    obj->somFree();
    return (0);
}
```

5) Compile and link the client program, as before.

6) Execute the client program, which now outputs the message:

```
> hello
 -- a Hello object at location 20062838 with msg: Hi There
```

Example 4 — Initializing a SOM Object

The previous example showed how to override the method **somPrintSelf**, introduced by **SOMObject**. As mentioned in that example, **somPrintSelf** should generally be overridden when implementing a new class. Another method introduced by **SOMObject** that should generally be overridden is **somDefaultInit**. The purpose of **somDefaultInit** is to provide a "default" *initializer* for the instance variables introduced by a class.

This example shows how to override **somDefaultInit** to give each "Hello" object's message an initial value when the object is first created. To learn more about initializers than shown in this example (including how to introduce new initializers that take arbitrary arguments, and how to explicitly invoke initializers) read Section 5.5, "Initializing and Uninitializing Objects," in Chapter 5, "Implementing Classes in SOM."

The overall process of overriding **somDefaultInit** is similar to that of the previous example. First, the IDL for "Hello" is modified. In addition to an **override** modifier, an **init** modifier is used to indicate that a stub procedure for an initialization method is desired (the stub procedures for initializers are different from those of normal methods).

1) Modify the interface declaration in "hello.idl."

```
#include <somobj.idl>

interface Hello : SOMObject
{
  void sayHello();

  attribute string msg;

#ifdef __SOMIDL__
implementation
{
    //# Method Modifiers:
    somPrintSelf: override;
    somDefaultInit: override, init;
};
#endif

};
```

2) Re-run the SOM Compiler on the updated hello.idl file, as before. The SOM Compiler extends the existing implementation file. Below is the initializer stub procedure that the SOM Compiler adds to the C language implementation file; C++ implementation files would be similarly revised:

```
SOM_Scope void SOMLINK
               somDefaultInit(Hello somSelf, somInitCtrl *ctrl)
{
  HelloData *somThis; /* set by BeginInitializer */
  somInitCtrl globalCtrl;
  somBooleanVector myMask;
  HelloMethodDebug("Hello", "somDefaultInit");
  Hello_BeginInitializer_somDefaultInit;

  Hello_Init_SOMObject_somDefaultInit(somSelf, ctrl);
  /*
   * local Hello initialization code added by programmer
   */
}
```

3) Customize the implementation.

   Here, the “msg” instance variable is set in the implementation template (rather than in the client program, as before). Thus, the “msg” is defined as part of the “Hello” object’s initialization.

```
SOM_Scope void SOMLINK
               somDefaultInit(Hello somSelf, somInitCtrl *ctrl)
{
  HelloData *somThis; /* set by BeginInitializer */
  somInitCtrl globalCtrl;
  somBooleanVector myMask;
  HelloMethodDebug(”Hello”, ”somDefaultInit”);
  Hello_BeginInitializer_somDefaultInit;

  Hello_Init_SOMObject_somDefaultInit(somSelf, ctrl);
  /*
   * local Hello initialization code added by programmer
   */
  __set_msg(somSelf, ”Initial Message”);
}
```

4) Update the client program to illustrate default initialization.

```
#include <hello.h>
main()
{
    Hello h = HelloNew();

    /* Execute the ”somPrintSelf” method */
    _somPrintSelf(h);
}
```

5) Compile and link the client program, as before.

6) Execute the client program:

```
> hello
-- a Hello object at 200633A8 with msg: Initial Message
```

Example 5 — Using Multiple Inheritance

The “Hello” class is useful for writing messages to the screen. So that clients can also write messages to printers and disk files, this example references two additional classes: “Printer” and “Disk”. The “Printer” class will manage messages to a printer, and the “Disk” class will manage messages sent to files. These classes can be defined as follows:

```
#include <somobj.idl>

interface Printer : SOMObject
{
    void stringToPrinter(in string s) ;
    // This method writes a string to a printer.
};

#include <somobj.idl>

interface Disk : SOMObject
{
    void stringToDisk(in string s) ;
    // This method writes a string to disk.

};
```

This example assumes the “Printer” and “Disk” classes are defined separately (in “print.idl” and “disk.idl”, for example), are implemented in separate files, and are linked with the other example code. Given the implementations of the “Printer” and “Disk” interfaces, the “Hello” class can use them by inheriting from them, as illustrated next.

1) Modify the interface declaration in “hello.idl”.

```
#include <disk.idl>
#include <printer.idl>

interface Hello : Disk, Printer
{
    void sayHello();

    attribute string msg;

    enum outputTypes {screen, printer, disk};
    // Declare an enumeration for the different forms of output

    attribute outputTypes output;
    // The current form of output

#ifdef __SOMIDL__

  implementation {
      somDefaultInit: override, init;
  };
#endif //#  __SOMIDL__
};
```

Notice that **SOMObject** is not listed as a parent of “Hello” above, because **SOMObject** is a parent of “Disk” (and of “Printer”).

The IDL specification above declares an enumeration “outputTypes” for the different forms of output, and an attribute “output” whose value will depend on where the client wants the output of the “sayHello” method to go.

Note: SOM IDL allows the use of structures, unions (though with a syntax different from C or C++), enumerations, constants, and typedefs, both inside and outside the body of an interface statement. Declarations that appear inside an interface body will be emitted in the header file (that is, in "hello.h" or "hello.xh"). Declarations that appear outside of an interface body do not appear in the header file (unless required by a special #pragma directive; see the SOM Compiler options in Chapter 4).

SOM IDL also supports all of the C and C++ preprocessor directives, including conditional compilation, macro processing, and file inclusion.

2) Re-run the SOM Compiler on the updated .idl file.

Unfortunately, when this is done, the implementation for **somDefaultInit** is not correctly updated to reflect the addition of two new parents to "Hello." This is because the implementation-file emitter never changes the bodies of existing method procedures. As a result, method procedures for initializer methods are *not* given new parent calls when the parents of a class are changed. One way to deal with this (when the parents of a class are changed) is to temporarily rename the method procedures for initializer methods, and then run the implementation emitter. Once this is done, the code in the renamed methods can be merged into the new templates, which will include all the appropriate parent method calls. When this is done here, the new implementation for **somDefaultInit** would appear as:

```
SOM_Scope void SOMLINK
               somDefaultInit(Hello somSelf, somInitCtrl *ctrl)
{
  HelloData *somThis; /* set by BeginInitializer */
  somInitCtrl globalCtrl;
  somBooleanVector myMask;
  HelloMethodDebug("Hello", "somDefaultInit");
  Hello_BeginInitializer_somDefaultInit;

  Hello_Init_Disk_somDefaultInit(somSelf, ctrl);
  Hello_Init_Printer_somDefaultInit(somSelf, ctrl);
  /*
   * local Hello initialization code added by programmer
   */
  __set_msg(somSelf, "Initial Message");
}
```

3) Continue to customize the implementation file, hello.c, as follows. Notice that the “sayHello” method (last discussed in Example 2) is now modified to allow alternate ways of outputting a “msg”.

```
SOM_Scope void  SOMLINK sayHello(Hello somSelf, Environment *ev)
{
    /* HelloData *somThis = HelloGetData(somSelf) ; */
    HelloMethodDebug(”Hello”,”sayHello”) ;

    switch (__get_output(somSelf, ev) ) {
            /*  for C++, use:  somSelf->_get_output(ev) */
    case Hello_screen:
        printf(”%s\n”, __get_msg(somSelf, ev) );
            /*  for C++, use:  somSelf->_get_msg(ev) */
        break;
    case Hello_printer:
        _stringToPrinter(somSelf, ev, __get_msg(somSelf, ev) );
            /* for C++, use:
             * somSelf->stringToPrinter(ev, somSelf->_get_msg(ev) );
             */
        break;
    case Hello_disk:
        _stringToDisk(somSelf, ev, __get_msg(somSelf, ev) );
            /*  for C++, use:
             *   somSelf->stringToDisk(ev, somSelf->_get_msg(ev) );
             */
        break;
    }
}
```

The “switch” statement invokes the appropriate method depending on the value of the “output” attribute. Notice how the “case” statements utilize the enumeration values of “outputTypes” declared in “hello.idl” by prefacing the enumeration names with the class name (Hello_screen, Hello_printer, and Hello_disk).

4) Update the client program, as illustrated next.

```
#include <hello.h>
        /* for C++, use ”hello.xh” and <stdio.h> */

int main(int argc, char *argv[])
{
    Hello a = HelloNew();
    Environment *ev = somGetGlobalEnvironment();

    /* Invoke ”sayHello” on an object and use each output */
    _sayHello(a, ev) ;        /* for C++, use:  a->sayHello(ev);  */
    __set_output(a, ev, Hello_printer) ;
                    /* C++:   a->_set_output(ev, Hello_printer); */
    _sayHello(a, ev) ;
    __set_output(a, ev, Hello_disk) ;
                    /* C++:   a->_set_output(ev, Hello_disk);  */
    _sayHello(a, ev) ;

    _somFree(a) ;  /* for C++, use: a->somFree(); */
    return (0);
}
```

5) Compile and link the client program as before, except also include the implementation files for the “Printer” and “Disk” classes in the compilation.

6) Execute the client program. Observe that the message that prints is the “msg” defined in Example 4 as part of the **somDefaultInit** initialization of the “Hello” object.

```
Initial Message
Initial Message          – goes to a Printer
Initial Message          – goes to Disk
```

This tutorial has described features of SOM IDL that will be useful to C and C++ programmers. SOM IDL also provides features such as full type checking, constructs for declaring private methods, and constructs for defining methods that receive and return pointers to structures. Chapter 4, “SOM IDL and the SOM Compiler,” gives a complete description of the SOM IDL syntax and also describes how to use the SOM Compiler. In addition, Chapter 5, “Implementing Classes in SOM,” provides helpful information for completing the implementation template, for using initializers (**somDefaultInit** or user-defined initialization methods), for defining SOM class libraries, and for customizing various aspects of SOMobjects execution.

Chapter 3. Using SOM Classes in Client Programs

Contents

Chapter 3. Using SOM Classes in Client Programs . . . . . 3 – 1
3.1 An Example Client Program . . . . . 3 – 3
3.2 Using SOM Classes — the Basics . . . . . 3 – 4
Declaring object variables . . . . . 3 – 4
Creating instances of a class . . . . . 3 – 5
Invoking methods on objects . . . . . 3 – 8
Making typical method calls . . . . . 3 – 8
Accessing attributes . . . . . 3 – 12
Using 'va_list' methods . . . . . 3 – 12
Using name-lookup method resolution . . . . . 3 – 14
A name-lookup example . . . . . 3 – 15
Obtaining a method's procedure pointer . . . . . 3 – 18
Method name or signature not known at compile time . . . . . 3 – 19
Using class objects . . . . . 3 – 20
Getting the class of an object . . . . . 3 – 20
Creating a class object . . . . . 3 – 20
Referring to class objects . . . . . 3 – 22
Compiling and linking . . . . . 3 – 23
3.3 Language-neutral Methods and Functions . . . . . 3 – 25
Generating output . . . . . 3 – 25
Getting information about a class . . . . . 3 – 25
Getting information about an object . . . . . 3 – 27
Methods . . . . . 3 – 27
Functions . . . . . 3 – 28
Debugging . . . . . 3 – 28
Checking the validity of method calls . . . . . 3 – 29
Exceptions and error handling . . . . . 3 – 29
Exception declarations . . . . . 3 – 30
Standard exceptions . . . . . 3 – 31
The Environment . . . . . 3 – 31
Setting an exception value . . . . . 3 – 32
Getting an exception value . . . . . 3 – 32
Example . . . . . 3 – 33
Memory management . . . . . 3 – 35
Using SOM equivalents to ANSI C functions . . . . . 3 – 35
Clearing memory for objects . . . . . 3 – 35
Clearing memory for the Environment . . . . . 3 – 36
SOM manipulations using somId's . . . . . 3 – 36

Chapter 3. Using SOM Classes in Client Programs

This chapter discusses how to use SOM classes that have already been fully implemented. That is, these topics describe the steps that a programmer uses to instantiate an object and invoke some method(s) on it from within an application program.

Who should read this chapter?

- Programmers who wish to use SOM classes that were originally developed by someone else will need to know the information in this chapter. These programmers often may not need the information from any subsequent chapters.

- By contrast, class implementors who are creating their own SOM classes should continue with Chapter 4, "SOM IDL and the SOM Compiler," and Chapter 5, "Implementing Classes in SOM," for complete information on the SOM Interface Definition Language (SOM IDL) syntax and other details of class implementation.

Programs that use a class are referred to as *client programs.* A client program can be written in C, in C++, or in another language. As noted, this chapter describes how client programs can use SOM classes (classes defined using SOM, as described in Chapter 2, "Tutorial for Implementing SOM Classes," in Chapter 4, "SOM IDL and the SOM Compiler," and in Chapter 5, "Implementing Classes in SOM"). Using a SOM class involves creating instances of a class, invoking methods on objects, and so forth. All of the methods, functions, and macros described here can also be used by class implementors within the implementation file for a class.

Note: "Using a SOM class," as described in this chapter, does *not* include subclassing the class in a client program. In particular, the C++ compatible SOM classes made available in the .xh binding file *cannot* be subclassed in C++ to create new C++ or SOM classes.

Some of the macros and functions described here are supplied as part of SOM's C and C++ *usage bindings*. These bindings are functions and macros defined in header files to be included in client programs. The usage bindings make it more convenient for C and C++ programmers to create and use instances of SOM classes. SOM classes can be also used without the C or C++ bindings, however. For example, users of other programming languages can use SOM classes, and C and C++ programmers can use a SOM class without using its language bindings. The language bindings simply offer a more convenient programmer's interface to SOM. Vendors of other languages may also offer SOM bindings; check with your language vendor for possible SOM support.

To use the C or C++ bindings for a class, a client program must include a header file for the class (using the **#include** preprocessor directive). For a C language client program, the file <*classFileStem*>.h must be included. For a C++ language client program, the file <*classFileStem*>.xh must be included. The SOM Compiler generates these header files from an IDL interface definition. The header files contain definitions of the macros and functions that make up the C or C++ bindings for the class. Whether the header files include bindings for the class's private methods and attributes (in addition to the public methods and attributes) depends on the IDL interface definition available to the user, and on how the SOM Compiler was invoked when generating bindings.

Usage binding headers automatically include any other bindings upon which they may rely. Client programs not using the C or C++ bindings for any particular class of SOM object (for example, a client program that does not know at compile time what classes it will be using)

should simply include the SOM-supplied bindings for **SOMObject**, provided in the header file “somobj.h” (for C programs) or “somobj.xh” (for C++ programs).

For each task that a user of a SOM class might want to perform, this chapter shows how the task would be accomplished by:

- a C programmer using the C bindings,
- a C++ programmer using the C++ bindings, or
- a programmer not using SOM’s C or C++ language bindings.

If neither of the first two approaches is applicable, the third approach can always be used.

3.1 An Example Client Program

Following is a C program that uses the class “Hello” (as defined in the Tutorial in Chapter 2). The “Hello” class provides one attribute, “msg”, of type string, and one method, “sayHello”. The “sayHello” method simply displays the value of the “msg” attribute of the object on which the method is invoked.

```
#include <hello.h>  /* include the header file for Hello */

int main(int argc, char *argv[])
{
    /* declare a variable (obj) that is a
     * pointer to an instance of the Hello class: */
    Hello obj;

    /* create an instance of the Hello class
     * and store a pointer to it in obj: */
    obj = HelloNew();

    /* invoke method _set_msg on obj with the argument
     * "Hello World Again". This method sets the value of
     * obj's 'msg' attribute to the specified string.
     */
    __set_msg(obj, somGetGlobalEnvironment(), "Hello World Again");

    /* invoke method sayHello on obj. This method prints
     * the value of obj's 'msg' attribute.  */
    _sayHello(obj, somGetGlobalEnvironment());

    _somFree(obj);
    return(0);
}
```

The C++ version of the foregoing client program is shown below:

```
#include <hello.xh>  /* include the header file for Hello */

int main(int argc, char *argv[])
{
    /* declare a variable (obj) that is a
     * pointer to an instance of the Hello class: */
    Hello *obj;

    /* create an instance of the Hello class
     * and store a pointer to it in obj: */
    obj = new Hello;

    /* invoke method _set_msg on obj with the argument
     * "Hello World Again". This method sets the value of
     * obj's 'msg' attribute to the specified string.  */
    obj->_set_msg(somGetGlobalEnvironment(), "Hello World Again");

    /* invoke method sayHello on obj. This method prints
     * the value of obj's 'msg' attribute.  */
    obj->sayHello(somGetGlobalEnvironment());

    obj->somFree();
    return(0);
}
```

These client programs both produce the output:

```
Hello World Again
```

3.2 Using SOM Classes — the Basics

This section describes the basic information needed to use SOM classes in a client program.

Declaring object variables

To declare an object variable, the name of an *object interface* defined in IDL is used as the *type* of the variable. The exact syntax is slightly different for C vs. C++ programmers. Specifically,

<*interfaceName*> obj ; in C programs or
<*interfaceName*> *obj ; in C++ programs

declares "obj" to be a pointer to an object that has type <*interfaceName*>. In SOM, objects of this type are instances of the SOM *class* named <*interfaceName*>, or of any SOM class derived from this class. Thus, for example,

```
Animal obj;          in C programs or
Animal *obj;         in C++ programs
```

declares "obj" as pointer to an object of type "Animal" that can be used to reference an instance of the SOM class "Animal" or any SOM class derived from "Animal". Note that the type of an object need not be the same as its class*;* an object of type "Animal" might not be an instance of the "Animal" class (rather, it might be an instance of some subclass of "Animal" — the "Cat" class, perhaps).

All SOM objects are of type **SOMObject**, even though they may not be instances of the **SOMObject** class. Thus, if it is not known at compile time what type of object the variable will point to, the following declaration can be used:

```
SOMObject obj;       in C programs or
SOMObject *obj;      in C++ programs.
```

Because the sizes of SOM objects are not known at compile time, instances of SOM classes must always be dynamically allocated. Thus, a variable declaration must always define a pointer to an object.

Note: in the C usage bindings, as within an IDL specification, an interface name used as a type *implicitly* indicates a pointer to an object that has that interface (this is required by the CORBA specification). The C usage bindings for SOM classes therefore hide the pointer with a C typedef for <*interfaceName*>. But this is not appropriate in the C++ usage bindings, which define a C++ class for <*interfaceName*>. Thus, it is not correct in C++ to use a declaration of the form:

<*interfaceName*> obj ; *not valid* in C++ programs

Note: If a C programmer also prefers to use explicit pointers to <*interfaceName*> types, then the SOM Compiler option **–maddstar** can be used when the C binding files are generated, and the explicit "*****" will then be required in declarations of object variables. (This option is required for compatibility with existing SOM OIDL code. For information on using the **–maddstar** option, see "Running the SOM Compiler" in Chapter 4, "SOM IDL and the SOM Compiler.")

Users of other programming languages must also define object variables to be pointers to the data structure used to represent SOM objects. The way this is done is programming-language dependent. The header file "somtypes.h" defines the structure of SOM objects for the C language.

Creating instances of a class

For C programmers with usage bindings, SOM provides the <*className*>**New** and the <*className*>**Renew** macros for creating instances of a class.

These macros are illustrated with the following two examples, each of which creates a single instance of class “Hello”:

```
obj = HelloNew();
obj = HelloRenew(buffer);
```

Using <*className*>New

After verifying that the <*className*> class object exists, the <*className*>**New** macro invokes the **somNew** method on the class object. This allocates enough space for a new instance of <*className*>, creates a new instance of the class, initializes this new object by invoking **somDefaultInit** on it, and then returns a pointer to it. The <*className*>**New** macro automatically creates the class object for <*className*>, as well as its ancestor classes and metaclass, if these objects have not already been created.

After a client program has finished using an object created using the <*className*>**New** macro, the object should be freed by invoking the method **somFree** on it :

```
_somFree(obj);
```

After uninitializing the object by invoking **somDestruct** on it, **somFree** calls the class object for storage deallocation. This is important because storage for an object created using the <*className*>**New** macro is allocated by the class of the object. Thus, only the class of the object can know how to reclaim the object’s storage.

Using <*className*>Renew

After verifying that the <*className*> class object exists, the <*className*>**Renew** macro invokes the **somRenew** method on the class object. <*className*>**Renew** is only used when the space for the object has been allocated previously. (Perhaps the space holds an old, uninitialized object that is not needed anymore.) This macro converts the given space into a new, initialized instance of <*className*> and returns a pointer to it. The programmer is responsible for ensuring that the argument of <*className*>**Renew** points to a block of storage large enough to hold an instance of class <*className*>. The SOM method **somGetInstanceSize** can be invoked on the class to determine the amount of memory required. Like <*className*>**New,** the <*className*>**Renew** macro automatically creates any required class objects that have not already been created.

Hint: When creating a large number of class instances, it may be more efficient to allocate at once enough memory to hold all the instances, and then invoke <*className*>**Renew** once for each object to be created, rather than performing separate memory allocations.

Using <*className*>NewClass

The C and C++ usage bindings for a SOM class also provide static linkage to a <*className*>**NewClass** function that can be used to create the class object. This can be useful if the class object is needed before its instances are created.

For example, the following C code uses the function **HelloNewClass** to create the “Hello” class object. The arguments to this function are defined by the usage bindings, and indicate the version of the class implementation that is assumed by the bindings. (For more detail on creation of classes, see the later topic, “Creating a class object.”) Once the class object has been created, the example invokes the method **somGetInstanceSize** on this class to determine the size of a “Hello” object, uses **SOMMalloc** to allocate storage, and then uses the **HelloRenew** macro to create ten instances of the “Hello” class:

```
#include <hello.h>
main()
{
SOMClass helloCls; /*  A pointer for the Hello class object */
Hello objA[10];    /*  an array of Hello instances */
unsigned char *buffer;
int i;
int size;

/* create the Hello class object:  */
helloCls = HelloNewClass(Hello_MajorVersion, Hello_MinorVersion);

/* get the amount of space needed for a Hello instance:
 * (somGetInstanceSize is a method provided by SOM.) */
size = _somGetInstanceSize(helloCls);
size = ((size+3)/4)*4;  /* round up to doubleword multiple */

/* allocate the total space needed for ten instances: */
buffer = SOMMalloc(10*size);

/* convert the space into ten separate Hello instances: */
for (i=0; i<10; i++)
   objA[i] = HelloRenew(buffer+i*size);
...
...
/* Uninitialize the objects and free them */
for (i=0; i<10; i++)
  _somDestruct(objA[i],0,0);
SOMFree(buffer);
}
```

When an object created with the <*className*>**Renew** macro is no longer needed, its storage must be freed using the dual to whatever method was originally used to allocate the storage. Two method pairs are typical:

- For example, if an object was originally initialized using the <*className*>**New** macro, then, as discussed previously, the client should use the **somFree** method on it.
- On the other hand, if the program uses the **SOMMalloc** function to allocate memory, as illustrated in the example above, then the **SOMFree** function must be called to free the objects' storage (because **SOMFree** is the dual to **SOMMalloc**). Before this is done, however, the objects in the region to be freed should be deinitialized by invoking the **somDestruct** method on them. This allows each object to free any memory that may have been allocated without the programmer's knowledge. (The **somFree** method also calls the **somDestruct** method.)

  Note: In the **somDestruct** method call above, the first zero indicates that memory should not be freed by the class of the object (that is, the programmer will do it explicitly). The second zero indicates that the class of the object is responsible for overall control of object uninitialization. For further discussion, see Section 5.5, "Initializing and Uninitializing Objects," in Chapter 5, "Implementing Classes in SOM."

For C++ programmers with usage bindings, instances of a class <*className*> can be created with a **new** operator provided by the usage bindings of each SOM class. The **new** operator automatically creates the class object for <*className*>, as well as its ancestor classes and metaclass, if they do not yet exist. After verifying the existence of the desired class object, the **new** operator then invokes the **somNewNoInit** method on the class. This allocates memory and creates a new instance of the class, but it does *not* initialize the new object.

Initialization of the new object is then performed using one of the C++ constructors defined by the usage bindings. (For further discussion, see Section 5.5, “Initializing and Uninitializing Objects,” in Chapter 5, “Implementing Classes in SOM.”) Two variations of the **new** operator require no arguments. When either is used, the C++ usage bindings provide a default constructor that invokes the **somDefaultInit** method on the new object. Thus, a new object initialized by **somDefaultInit** would be created using either of the forms:

new <*className*>

new <*className*>()

For example:

```
obj = new Hello;

obj1 = new Hello();
```

For convenience, pointers to SOM objects created using the **new** operator can be freed using the **delete** operator, just as for normal C++ objects (or, the **somFree** method could be used):

```
delete obj;

obj1->somFree;
```

When previously allocated space will be used to hold a new object, C++ programmers should use the **somRenew** method, described below. C++ bindings do not provide a macro for this purpose.

somNew and somRenew: C and C++ programmers, as well as programmers using other languages, can create instances of a class using the SOM methods **somNew** and **somRenew**, invoked on the class object. As discussed and illustrated above for the C bindings, the class object must first be created using the <*className*>**NewClass** procedure (or, perhaps, using the **somFindClass** method — see the section “Using class objects” later in this chapter).

The **somNew** method invoked on the class object creates a new instance of the class, initializes the object using **somDefaultInit**, and then returns a pointer to the new object. For instance, the following C example creates a new object of the “Hello” class.

```
#include <hello.h>
main()
{
  SOMClass helloCls;   /* a pointer to the Hello class */
  Hello obj;           /* a pointer to a Hello instance */

  /* create the Hello class  */
  helloCls = HelloNewClass(Hello_MajorVersion, Hello_MinorVersion);
  obj = _somNew(helloCls); /* create the Hello instance */
}
```

An object created using the **somNew** method should be freed by invoking the **somFree** method on it after the client program is finished using the object.

The **somRenew** method invoked on the class object creates a new instance of a class using the given space, rather than allocating new space for the object. The method converts the given space into an instance of the class, initializes the new object using **somDefaultInit**, and then returns a pointer to the new object. The argument to **somRenew** must point to a block of storage large enough to hold the new instance. The method **somGetInstanceSize** can be used to determine the amount of memory required. For example, the following C++ code creates ten instances of the “Hello” class:

```
#include <hello.xh>
#include <somcls.xh>
main()
{
  SOMClass *helloCls; // a pointer to the Hello class
  Hello *objA[10];    //  an array of Hello instance pointers
  unsigned char *buffer;
  int i;
  int size;

// create the Hello class object
helloCls = HelloNewClass(Hello_MajorVersion, Hello_MinorVersion);

// get the amount of space needed for a Hello instance:
size = helloCls->somGetInstanceSize();
size = ((size+3)/4)*4;  // round up to doubleword multiple

// allocate the total space needed for ten instances
buffer = SOMMalloc(10*size);


// convert the space into ten separate Hello objects
for (i=0; i<10; i++)
   objA[i] = helloCls->somRenew(buffer+i*size);

 ...

// Uninitialize the objects and free them
 for (i=0; i<10; i++)
   objA[i]->somDestruct(0,0);
 SOMFree(buffer);
}
```

The **somNew** and **somRenew** methods are useful for creating instances of a class when the header file for the class is not included in the client program at compile time. (The name of the class might be specified by user input, for example.) However, the <*className*>**New** macro (for C) and the **new** operator (for C++) can *only* be used for classes whose header file is included in the client program at compile time.

Objects created using the **somRenew** method should be freed by the client program that allocated it, using the dual to whatever allocation approach was initially used. If the **somFree** method is not appropriate (because the **somNew** method was not initially used), then, before memory is freed, the object should be explicitly deinitialized by invoking the **somDestruct** method on it. (The **somFree** method calls the **somDestruct** method. Refer to the previous C example for **Renew** for an explanation of the arguments to **somDestruct**.)

Invoking methods on objects

This topic describes the general way to invoke methods in C/C++ and in other languages, and then presents subtopics for more specialized situations.

Making typical method calls

For C programmers with usage bindings: To invoke a method in C, use the macro:

_<*methodName*> (*receiver, args)*

(that is, an underscore followed by the method name). Arguments to the macro are the receiver of the method followed by all of the arguments to the method. For example:

```
_foo(obj, somGetGlobalEnvironment(), x, y);
```

This invokes method “foo” on “obj” (the remaining arguments are arguments to the method “foo”). This expression can be used anywhere that a standard function call can be used in C.

Required arguments

In C, calls to methods defined using IDL require at least two arguments — a pointer to the *receiving object* (the object responding to the method) and a value of type (**Environment ***). The **Environment** data structure is specified by CORBA, and is used to pass environmental information between a caller and a called method. For example, it is used to return exceptions. (For more information on how to supply and use the **Environment** structure, see the section entitled "Exceptions and error handling" later in this chapter.)

In the IDL definition of a method, by contrast, the receiver and the **Environment** pointer are *not* listed as parameters to the method. (Unlike the receiver, the **Environment** pointer is considered a method parameter, even though it is never explicitly specified in IDL. For this reason, it is called an *implicit* method parameter.) For example, if a method is defined in a .idl file with two parameters, as in:

```
int foo (in char c, in float f);
```

then, with the C usage bindings, the method would be invoked with four arguments, as in:

```
intvar = _foo(obj, somGetGlobalEnvironment(), x, y);
```

where "obj" is the object responding to the method and "x" and "y" are the arguments corresponding to "c" and "f", above.

If the IDL specification of the method includes a *context* specification, then the method has an additional (implicit) **context** parameter. Thus, when invoking the method, this argument must follow immediately after the **Environment** pointer argument. (None of the SOM-supplied methods require **context** arguments.) The **Environment** and **context** method parameters are prescribed by the CORBA standard.

If the IDL specification of the class that introduces the method includes the **callstyle=oidl** modifier, then the (**Environment ***) and **context** arguments should not be supplied when invoking the method. That is, the receiver of the method call is followed immediately by the arguments to the method (if any). Some of the classes supplied in the SOMobjects Toolkit (including **SOMObject, SOMClass**, and **SOMClassMgr**) are defined in this way, to ensure compatibility with the previous release of SOM. The *SOMobjects Developer Toolkit Programmers Reference Manual* specifies for each method whether these arguments are used.

If you use a C expression to compute the first argument to a method call (the receiver), you must use an expression without side effects, because the first argument is evaluated twice by the **_**<*methodName*> macro expansion. In particular, a **somNew** method call or a macro call of <*className*>**New** can *not* be used as the first argument to a C method call, because doing so would create two new class instances rather than one.

Following the initial, required arguments to a method (the receiving object, the **Environment**, if any, and the **context**, if any), you enter any additional arguments required by that method, as specified in IDL. For a discussion of how IDL **in**/**out**/**inout** argument types map to C/C++ data types, see the topic "Parameter list" in Chapter 4, "SOM IDL and the SOM Compiler."

Short form vs long form

If a client program uses the bindings for two different classes that introduce or inherit two different methods of the same name, then the _<*methodName*> macro described above (called the *short form*) will not be provided by the bindings, because the macro would be ambiguous in that circumstance. The following *long form* macro, however, is always provided by the usage bindings for each class that supports the method:

<*className*>_<*methodName*> (*receiver, args*)

For example, method "foo" supported by class "Bar" can be invoked as:

```
Bar_foo(obj, somGetGlobalEnvironment(), x, y)        (in C)
```

where "obj" has type "Bar" and "x" and "y" are the arguments to method "foo".

In most cases (where there is no ambiguity, and where the method is not a **va_list** method, as described in the subsequent subtopic "Using 'va_list' methods"), a C programmer may use either the short or the long form of a method invocation macro interchangeably. However, *only* the long form complies with the CORBA standard for C usage bindings. If you wish to write code that can be easily ported to other vendor platforms that support the CORBA standard, use the long form exclusively. The long form is always available for every method that a class supports. The short form is provided both as a programming convenience and for source code compatibility with release 1 of SOM.

In order to use the long form, a programmer will usually know what type an object is expected to have. If this is not known, but the different methods have the same signature, the method can be invoked using name-lookup resolution, as described in a following subtopic of this section.

For C++ programmers with usage bindings: To invoke a method, use the standard C++ form shown below:

obj–><*methodName*> (*args*)

where *args* are the arguments to the method. For instance, the following example invokes method "foo" on "obj":

```
obj->foo(somGetGlobalEnvironment(), x, y)
```

Required arguments

All methods introduced by classes declared using IDL (except those having the SOM IDL **callstyle=oidl** modifier) have at least one parameter — a value of type (**Environment ***). The **Environment** data structure is used to pass environmental information between a caller and a called method. For example, it is used to return exceptions. For more information on how to supply and use the **Environment** structure, see the section entitled "Exceptions and error handling" later in this chapter.

The **Environment** pointer is an implicit parameter. That is, in the IDL definition of a method, the **Environment** pointer is *not* explicitly listed as a parameter to the method. For example, if a method is defined in IDL with two explicit parameters, as in:

```
int foo (in char c, in float f);
```

then the method would be invoked from C++ bindings with three arguments, as in:

```
intvar = obj->foo(somGetGlobalEnvironment(), x, y);
```

where "obj" is the object responding to the method and "x" and "y" are the arguments corresponding to "c" and "f", above.

If the IDL specification of the method includes a *context* specification, then the method has a second implicit parameter, of type **context**, and the method must be invoked with an additional **context** argument. This argument must follow immediately after the **Environment** pointer argument. (No SOM-supplied methods require **context** arguments.) The **Environment** and **context** method parameters are prescribed by the CORBA standard.

If the IDL specification of the class that introduces the method includes the **callstyle=oidl** modifier, then the (**Environment ***) and **context** arguments should not be supplied when the method is invoked. Some of the classes supplied in the SOMobjects Toolkit (including **SOMObject, SOMClass**, and **SOMClassMgr**) are defined in this way, to ensure compatibility with the previous release of SOM. The *SOMobjects Developer Toolkit Programmers Reference Manual* specifies for each method whether these arguments are used.

Following the initial, required arguments to a method (the receiving object, the **Environment**, if any, and the **context**, if any), you enter any additional arguments required by that method, as specified in IDL. For a discussion of how IDL **in**/**out**/**inout** argument types map to C/C++ data types, see the topic "Parameter list" in Chapter 4, "SOM IDL and the SOM Compiler."

For non-C/C++ programmers: To invoke a *static method* (that is, a method declared when defining an OIDL or IDL object interface) without using the C or C++ usage bindings, a programmer can use the **somResolve** procedure. The **somResolve** procedure takes as arguments a pointer to the object on which the method is to be invoked and a *method token* for the desired method. It returns a pointer to the method's procedure (or raises a fatal error if the object does not support the method). Depending on the language and system, it may be necessary to cast this procedure pointer to the appropriate type; the way this is done is language-specific.

The method is then invoked by calling the procedure returned by **somResolve**, passing the method's receiver, the **Environment** pointer (if necessary), the **context** argument (if necessary) and the remainder of the method's arguments, if any. Note that the means for calling a procedure, given a pointer returned by **somResolve**, is language-specific. (See the section above for C programmers. The arguments to a method procedure are the same as the arguments passed using the long form of the C-language method-invocation macro for that method.)

Using **somResolve** requires the programmer to know where to find the *method token* for the desired method. Method tokens are available from class objects that support the method (via the method **somGetMethodToken**), or from a global data structure, called the *ClassData structure,* corresponding to the class that introduces the method. In C and C++ programs with access to the definitions for ClassData structures provided by usage bindings, the method token for method *methodName* introduced by class *className* may be accessed by the following expression:

<*className*>**ClassData.**<*methodName*>

For example, the method token for method "sayHello" introduced by class "Hello" is stored at location `HelloClassData.sayHello`, for C and C++ programmers. The way method tokens are accessed in other languages is language-specific.

As an example of using offset resolution to invoke methods from a programming language other than C/C++, one would do the following to create an instance of a SOM Class *X* in Smalltalk:

1. Initialize the SOM run-time environment, if it has not previously been initialized, using the **somEnvironmentNew** function.
2. If the class object for class *X* has not yet been created, use **somResolve** with arguments **SOMClassMgrObject** (returned by **somEnvironmentNew** in step 1) and the method token for the **somFindClass** method, to obtain a method procedure pointer for the **somFindClass** method. Use the method procedure for **somFindClass** to create the class object for class *X*: Call the procedure with arguments **SOMClassMgrObject**, the result of calling the **somIdFromString** function with argument *"X",* and the major and minor version numbers for class *X* (or zero). The procedure returns the class object for class *X*.
3. Use **somResolve** with arguments representing the class object for *X* (returned by **somFindClass** in step 2) and the method token for the **somNew** method, to obtain a method procedure pointer for method **somNew.** (The **somNew** method is used to create instances of class *X*.)
4. Call the method procedure for **somNew** (using the method procedure pointer obtained in step 3) with the class object for *X* (returned by **somFindClass** in step 3) as the argument. The procedure returns a new instance of class *X*.

In addition to **somResolve**, SOM also supplies the **somClassResolve** procedure. Instead of an object, the **somClassResolve** procedure takes a class as its first argument, and then selects a method procedure from the instance method table of the passed class. (The **somResolve** procedure, by contrast, selects a method procedure from the instance method table of the class of which the passed object is an instance.) The **somClassResolve** procedure therefore supports *casted* method resolution. See the *SOMobjects Developer Toolkit Programmers Reference Manual* for more information on **somResolve** and **somClassResolve**.

If the programmer does not know at compile time which class introduces the method to be invoked, or if the programmer cannot directly access method tokens, then the procedure **somResolveByName** can be used to obtain a method procedure using name-lookup resolution, as described in the next section.

If the signature of the method to be invoked is not known at compile time, but can be discovered at run time, use **somResolve** or **somResolveByName** to get a pointer to the **somDispatch** method procedure, then use it to invoke the specific method, as described below under "Method name or signature not known at compile time."

Accessing attributes

In addition to methods, SOM objects can also have attributes. An *attribute* is an IDL shorthand for declaring methods, and does not necessarily indicate the presence of any particular instance data in an object of that type. Attribute methods are called "get" and "set" methods. For example, if a class "Hello" declares an attribute called "msg", then object variables of type "Hello" will support the methods **_get_msg** and **_set_msg** to access or set the value of the "msg" attribute. (Attributes that are declared as "readonly" have no "set" method, however.)

The "get" and "set" methods are invoked in the same way as other methods. For example, given class "Hello" with attribute "msg" of type **string**, the following code segments set and get the value of the "msg" attribute:

For C:

```
#include <hello.h>
Hello obj;
Environment *ev = somGetGlobalEnvironment();

obj = HelloNew();
__set_msg(obj, ev, "Good Morning");/*note:two leading underscores */
printf("%s\n", __get_msg(obj, ev));
```

For C++:

```
#include <hello.xh>
#include <stdio.h>
Hello *obj;
Environment *ev = somGetGlobalEnvironment();

obj = new Hello;
obj->_set_msg(ev, "Good Morning");
printf("%s\n", obj->_get_msg(ev));
```

Attributes available with each class, if any, are described in the documentation of the class itself in the *SOMobjects Developer Toolkit Programmers Reference Manual.*

Using 'va_list' methods

SOM supports methods whose final argument is a **va_list**. A va_list is a datatype whose representation depends on the operating system platform. On AIX, OS/2 and Windows, a va_list is simply a pointer to a block of memory that contains a number of arguments, sequentially laid out in memory. In the future, SOMobjects may provide special facilities for creating a va_list, but on the above systems the C/C++ **va_arg** macro can be used for this purpose. This is illustrated below.

As a convenience, methods whose final argument is a va_list, such as:

```
void setMany(in short start, in short numArgs, in va_list ap);
```

can be invoked from C and C++ by specifying a variable number of arguments, as follows:

For C:

```
_setMany(aVector, somGetGlobalEnvironment(), 2, 4, 20, 12, 32, 41);
```

For C++:

```
aVector->setMany(somGetGlobalEnvironment(), 2, 4, 20, 12, 32, 41);
```

C programmers must be aware that the short form of the invocation macro used above to pass a variable number of arguments to a va_list method is only available in the absence of ambiguity. The long-form macro (which is always available) requires an explicit va_list argument, as described next.

Note: As mentioned above, the short form may not be available due to ambiguity in the bindings. If a variable-argument interface is desired in such cases, however, you can inspect the usage bindings to find the short form expansion and make use of this directly.

As an alternative to the preceding example, both C and C++ programmers can use an explicit va_list argument, as suggested by the method definition. That is, to use a va_list argument, a method must be invoked as <*className*>_<*methodName*>, where <*className*> is the name of a class that supports the method.

For example, assume that class “Vector” supports the “setMany” method above. The following code first constructs a variable of type va_list, uses the **va_arg** macro to store the arguments to “setMany” in the va_list, and then invokes the “setMany” method with this variable:

For C:

```
va_list start_ap, ap;
Vector aVector = VectorNew();
...
start_ap = ap = (va_list) SOMMalloc(4 * sizeof(long));
va_arg(ap, long) = 20;
va_arg(ap, long) = 12;
va_arg(ap, long) = 32;
va_arg(ap, long) = 41;
Vector_setMany(aVector, somGetGlobalEnvironment(), 2, 4, start_ap);
```

For C++:

```
va_list start_ap, ap;
Vector *aVector = new Vector;
...
start_ap = ap = (va_list) SOMMalloc(4 * sizeof(long));
va_arg(ap, long) = 20;
va_arg(ap, long) = 12;
va_arg(ap, long) = 32;
va_arg(ap, long) = 41;
aVector->Vector_setMany(somGetGlobalEnvironment(), 2, 4, start_ap);
```

Using name-lookup method resolution

For C/C++ programmers: Offset resolution is the most efficient way to select the method procedure appropriate to a given method call. Client programs can, however, invoke a method using "name-lookup" resolution instead of offset resolution. The C and C++ bindings for method invocation use offset resolution by default, but methods defined with the **namelookup** SOM IDL modifier result in C bindings in which the short form invocation macro uses name-lookup resolution instead. Also, for both C and C++ bindings, a special *lookup_<methodName>* macro is defined.

Name-lookup resolution is appropriate in the case where a programmer knows at compile time which arguments will be expected by a method (that is, its *signature*), but does not know the *type* of the object on which the method will be invoked. For example, name-lookup resolution can be used when two different classes introduce different methods of the same name and signature, and it is not known which method should be invoked (because the type of the object is not known at compile time).

Name-lookup resolution is also used to invoke *dynamic methods* (that is, methods that have been added to a class's interface at run time rather than being specified in the class's IDL specification). For more information on name-lookup method resolution, see the topic "Method Resolution" in Chapter 5, "Implementing Classes in SOM."

For C: To invoke a method using name-lookup resolution, when using the C bindings for a method that has been implemented with the **namelookup** modifier, use either of the following macros:

_<*methodName*> (*receiver, args)*

lookup_<*methodName*> (*receiver, args)*

Thus, the short-form method invocation macro results in name-lookup resolution (rather than offset resolution), when the method has been defined as a **namelookup** method. (The long form of the macro for offset resolution is still available in the C usage bindings.) If the method takes a variable number of arguments, then the first form shown above is used when supplying a variable number of arguments, and the second form is used when supplying a **va_list** argument in place of the variable number of arguments.

For C++: To invoke a method using name-lookup resolution, when using the C++ bindings for a method that has been defined with the **namelookup** modifier, use either of the following macros:

lookup_<*methodName*> (*receiver, args)*

<*className*>**_lookup**_<*methodName*> (*receiver, args)*

If the method takes a variable number of arguments, then the first form shown above is used when supplying a variable number of arguments, and the second form is used when supplying a **va_list** argument in place of the variable number of arguments. Note that the offset-resolution forms for invoking methods using the C++ bindings are also still available, even if the method has been defined as a **namelookup** method.

For C/C++: To invoke a method using name-lookup resolution, when the method has *not* been defined as a **namelookup** method:

- Use the **somResolveByName** procedure (described in the following section)**,** or any of the methods **somLookupMethod, somFindMethod** or **somFindMethodOk** to obtain a pointer to the procedure that implements the desired method.
- Then, invoke the desired method by calling that procedure, passing the method's intended receiver, the **Environment** pointer (if needed), the **context** argument (if needed), and the remainder of the method's arguments, if any.

The **somLookupMethod, somFindMethod** and **somFindMethodOK** methods are invoked on a class object (the class of the method receiver should be used), and take as an argument the **somId** for the desired method (which can be obtained from the method's name using the **somIdFromString** function). For more information on these methods, see the *SOMobjects Developer Toolkit Programmers Reference Manual*.

Important Note: SOM provides many ways for a SOM user to acquire a pointer to a method procedure. Once this is done, it becomes the user's responsibility to make appropriate use of this procedure.

- First, the procedure should only be used on objects for which this is appropriate. Otherwise, run-time errors are likely to result.
- Second, when the procedure is used, it is essential that the compiler be given correct information concerning the signature of the method and the linkage required by the method. (On many systems, there are different ways to pass method arguments, and linkage information tells a compiler how to pass the arguments indicated by a method's signature).

SOM method procedures on OS/2 must be called with "system" linkage. On Windows, method procedures use the C linkage convention. On AIX, there is only one linkage convention for procedure calls. While C and C++ provide standard ways to indicate a method signature, the way to indicate linkage information depends on the specific compiler and system. For each method declared using OIDL or IDL, the C and C++ usage bindings therefore use conditional macros and a typedef to name a type that has the correct linkage convention. This type name can then be used by programmers with access to the usage bindings for the class that introduces the method whose procedure pointer is used. The type is named **somTD**_<*className*>_<*methodName*>. This is illustrated in the following example, and further details are provided in the section below, entitled "Obtaining a method's procedure pointer."

A name-lookup example

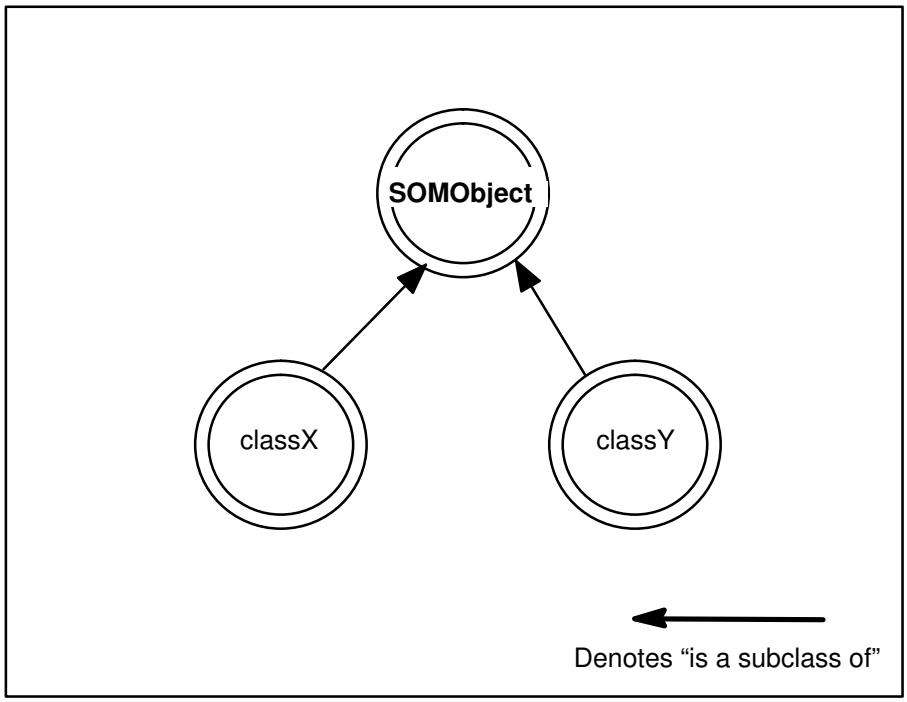
The following example shows the use of name-lookup by a SOM client programmer. Name-lookup resolution is appropriate when a programmer knows that an object will respond to a method of some given name, but does not know enough about the type of the object to use offset method resolution. How can this happen? It normally happens when a programmer wants to write generic code, using methods of the same name and signature that are applicable to different classes of objects, and yet these classes have no common ancestor that introduces the method. This can easily occur in single-inheritance systems (such as Smalltalk and SOM release 1) and can also happen in multiple-inheritance systems such as SOM release 2 — when class hierarchies designed by different people are brought together for clients' use.

If multiple inheritance is available, it is always possible to create a common class ancestor into which methods of this kind can be migrated. A refactoring of this kind often implements a semantically pleasing generalization that unifies common features of two previously unrelated class hierarchies. This step is most practical, however, when it does not require the redefinition or recompilation of current applications that use offset resolution. SOM is unique in that it allows this.

However, such refactoring must redefine the classes that originally introduced the common methods (so the methods can be inherited from the new "unifying" class instead). A client programmer who simply wants to create an application may not control the implementations of the classes. Thus, the use of name-lookup method resolution seems the best alternative for programmers who do not want to define new classes, but simply to make use of available ones.

For example, assume the existence of two different SOM classes, "classX" and "classY", whose only common ancestor is **SOMObject**, and who both introduce a method named "reduce" that accepts a *string* as an argument and returns a *long*. We assume that the classes were not

designed in conjunction with each other. As a result, it is unlikely that the “reduce” method was defined with a **namelookup** modifier. The following figure illustrates the class hierarchy for this example.



Following is a C++ generic procedure that uses name-lookup method resolution to invoke the “reduce” method on its argument, which may be either of type “classX” or “classY”. Note that there is no reason to include classY’s usage bindings, since the typedef provided for the “reduce” method procedure in “classX” is sufficient for invoking the method procedure, independently of whether the target object is of type “classX” or “classY”.

```
#include <classX.xh> // use classX's method proc typedef

// this procedure can be invoked on a target of type
// classX or classY.

long generic_reduce1(SOMObject *target, string arg)
{
   somTD_classX_reduce reduceProc = (somTD_classX_reduce)
        somResolveByName(target, "reduce");
   return reduceProc(target, arg);
}
```

On the other hand, if the classes were designed in conjunction with each other, and the class designer felt that programmers might want to write generic code appropriate to either class of object, the **namelookup** modifier might have been used. This is a possibility, even with multiple inheritance. However, it is much more likely that the class designer would use multiple inheritance to introduce the “reduce” method in a separate class, and then use this other class as a parent for both “classX” and “classY” (thereby allowing the use of offset resolution).

In any case, if the “reduce” method in “classX” were defined as a **namelookup** method, the following code would be appropriate. Note that the name-lookup support provided by “classX” usage bindings is still appropriate for use on targets that do not have type “classX”. As a result,

the “reduce” method introduced by “classY” need not have been defined as a **namelookup** method.

```
#include <classX.xh> // use classX's name-lookup support

// this procedure can be invoked on a target of type
// classX or classY.

long generic_reduce2(SOMObject *target, string arg)
{
   return lookup_reduce(target, arg);
}
```

For non-C/C++ programmers: Name-lookup resolution is useful for non-C/C++ programmers when the type of an object on which a method must be invoked is not known at compile time or when method tokens cannot be directly accessed by the programmer. To invoke a method using name-lookup resolution when not using the C or C++ usage bindings, use the **somResolveByName** procedure to acquire a procedure pointer. How the programmer indicates the method arguments and the linkage convention in this case is compiler specific.

The **somResolveByName** procedure takes as arguments a pointer to the object on which the method is to be invoked and the name of the method, as a string. It returns a pointer to the method's procedure (or NULL if the method is not supported by the object). The method can then be invoked by calling the method procedure, passing the method's receiver, the **Environment** pointer (if necessary), the **context** argument (if necessary), and the rest of the method's arguments, if any. (See the section above for C programmers; the arguments to a method procedure are the same as the arguments passed to the long-form C-language method-invocation macro for that method.)

As an example of invoking methods using name-lookup resolution using the procedure **somResolveByName**, the following steps are used to create an instance of a SOM Class *X* in Smalltalk:

1. Initialize the SOM run-time environment (if it is not already initialized) using the **somEnvironmentNew** function.

2. If the class object for class *X* has not yet been created, use **somResolveByName** with the arguments **SOMClassMgrObject** (returned by **somEnvironmentNew** in step 1) and the string *“somFindClass”,* to obtain a method procedure pointer for the **somFindClass** method. Use the method procedure for **somFindClass** to create the class object for class *X*: Call the method procedure with these four arguments: **SOMClassMgrObject**; the variable holding class *X*'s **somId** (the result of calling the **somIdFromString** function with argument *“X”*); and the major and minor version numbers for class *X* (or zero). The result is the class object for class *X*.

3. Use **somResolveByName** with arguments the class object for *X* (returned by **somFindClass** in step 2) and the string *“somNew”*, to obtain a method procedure pointer for method **somNew.** (This **somNew** method is used to create instances of a class.)

4. Call the method procedure for **somNew** (using the method procedure pointer obtained in step 3) with the class object for *X* (returned by **somFindClass** in step 3) as the argument. The result is a new instance of class *X.* How the programmer indicates the method arguments and the linkage convention is compiler-specific.

Obtaining a method's procedure pointer

Method resolution is the process of obtaining a pointer to the procedure that implements a particular method for a particular object at run time. The method is then invoked subsequently by calling that procedure, passing the method's intended receiver, the **Environment** pointer (if needed), the **context** argument (if needed), and the method's other arguments, if any. C and C++ programmers may wish to obtain a pointer to a method's procedure for efficient repeated invocations.

Obtaining a pointer to a method's procedure is achieved in one of two ways, depending on whether the method is to be resolved using **offset** resolution or **name-lookup** resolution. Obtaining a method's procedure pointer via offset resolution is faster, but it requires that the name of the class that introduces the method and the name of the method be known at compile time. It also requires that the method be defined as part of that class's interface in the IDL specification of the class. (See the topic "Method Resolution" in Chapter 5, "Implementing Classes in SOM," for more information on offset and name-lookup method resolution.)

Offset resolution

To obtain a pointer to a procedure using **offset** resolution, the C/C++ usage bindings provide the **SOM_Resolve** and **SOM_ResolveNoCheck** macros. The usage bindings themselves use the first of these, **SOM_Resolve**, for offset-resolution method calls. The difference in the two macros is that the **SOM_Resolve** macro performs consistency checking on its arguments, but the macro **SOM_ResolveNoCheck**, which is faster, does not. Both macros require the same arguments:

SOM_Resolve(<*receiver*>, <*className*>, <*methodName*>)
SOM_ResolveNoCheck(<*receiver*>, <*className*>, <*methodName*>)

where the arguments are as follows:

receiver	— The object to which the method will apply. It should be specified as an expression without side effects.
className	— The name of the class that introduces the method.
methodName	— The name of the desired method.

These two names (*className* and *methodName*) must be given as tokens, rather than strings or expressions. (For example, as Animal rather than "Animal".) If the symbol **SOM_TestOn** is defined and the symbol **SOM_NoTest** is not defined in the current compilation unit, then **SOM_Resolve** verifies that *receiver* is an instance of *className* or some class derived from *className*. If this test fails, an error message is output and execution is terminated.

The **SOM_Resolve** and **SOM_ResolveNoCheck** macros use the procedure **somResolve** to obtain the entry-point address of the desired method procedure (or raise a fatal error if *methodName* is not introduced by *className)*. This result can be directly applied to the method arguments, or stored in a variable of generic procedure type (for example, **somMethodPtr**) and retained for later method use. This second possibility would result in a loss of information, however, for the reasons now given.

The **SOM_Resolve** or **SOM_ResolveNoCheck** macros are especially useful because they cast the method procedure they obtain to the right type to allow the C or C++ compiler to call this procedure with *system linkage* and with the appropriate arguments. This is why the result of **SOM_Resolve** is immediately useful for calling the method procedure, and why storing the result of **SOM_Resolve** in a variable of some "generic" procedure type results in a loss of information. The correct type information can be regained, however, because the type used by **SOM_Resolve** for casting the result of **somResolve** is available from C/C++ usage bindings using the typedef name **somTD_**<*className*>_<*methodName*>. This type name describes a

pointer to a method procedure for *methodName* introduced by class *className*. If the final argument of the method is a **va_list**, then the method procedure returned by **SOM_Resolve** or **SOM_ResolveNoCheck** must be called with a **va_list** argument, and not a variable number of arguments.

Below is a C example of using **SOM_Resolve** to obtain a method procedure pointer for method "sayHello", introduced by class "Hello", and using it to invoke the method on "obj." (Assume that the only argument required by the "sayHello" method is the **Environment** pointer.)

```
somMethodProc *p;
SOMObject obj = HelloNew();
p = SOM_Resolve(obj, Hello, sayHello);
((somTD_Hello_sayHello)p) (obj, somGetGlobalEnvironment());
```

SOM_Resolve and **SOM_ResolveNoCheck** can only be used to obtain method procedures for *static methods* (methods that have been declared in an IDL specification for a class) and not methods that are added to a class at run time. See the *SOMobjects Programmers Reference Manual* for more information and examples on **SOM_Resolve** and **SOM_ResolveNoCheck**.

Name-lookup method resolution

To obtain a pointer to a method's procedure using **name-lookup** resolution, use the **somResolveByName** procedure (described in the following section)**,** or any of the **somLookupMethod, somFindMethod** and **somFindMethodOK** methods. These methods are invoked on a class object that supports the desired method, and they take an argument specifying the a **somId** for the desired method (which can be obtained from the method's name using the **somIdFromString** function). For more information on these methods and for examples of their use, see the *SOMobjects Developer Toolkit Programmers Reference Manual*.

Method name or signature not known at compile time

If the programmer does not know a method's name at compile time (for example, it might be specified by user input), then the method can be invoked in one of two ways, depending upon whether its signature is known:

- Suppose the signature of the method is known at compile time (even though the method name is not). In that case, when the name of the method becomes available at run time, the **somLookupMethod, somFindMethod** or **somFindMethodOk** methods or the **somResolveByName** procedure can be used to obtain a pointer to the method's procedure using name-lookup method resolution, as described in the preceding topics. That method procedure can then be invoked, passing the method's intended receiver, the **Environment** pointer (if needed), the **context** argument (if needed), and the remainder of the method's arguments.
- If the method's signature is unknown until run time, then dispatch-function resolution is indicated, as described in the next topic.

Dispatch-function method resolution

If the *signature* of the method is *not* known at compile time (and hence the method's argument list cannot be constructed until run time), then the method can be invoked at run time by (a) placing the arguments in a variable of type **va_list** at run time and (b) either using the **somGetMethodData** method followed by use of the **somApply** function, or by invoking the **somDispatch** or **somClassDispatch** method. Using **somApply** is more efficient, since this is what the **somDispatch** method does, but it requires two steps instead of one. In either case, the result invokes a "stub" procedure called an *apply stub*, whose purpose is to remove the method arguments from the **va_list**, and then pass them to the appropriate method procedure in the way expected by that procedure. For more information on these methods and for examples of their use, see the **somApply** function, and the **somGetMethodData, somDispatch,** and **somClassDispatch** methods in the *SOMobjects Programmers Reference Manual*.

Using class objects

Using a class object encompasses three aspects: getting the class of an object, creating a new class object, or simply referring to a class object through the use of a pointer.

Getting the class of an object

To get the class that an object is an instance of, SOM provides a method called **somGetClass**. The **somGetClass** method takes an object as its only argument and returns a pointer to the class object of which it is an instance. For example, the following statements store in “myClass” the class object of which “obj” is an instance.

```
myClass = _somGetClass(obj);   (for C)

myClass = obj->somGetClass();   (for C++)
```

Getting the class of an object is useful for obtaining information about the object; in some cases, such information cannot be obtained directly from the object, but only from its class. The section below entitled “Getting information about a class” describes the methods that can be invoked on a class object after it is obtained using **somGetClass**.

The **somGetClass** method can be overridden by a class to provide enhanced or alternative semantics for its objects. Because it is usually important to respect the intended semantics of a class of objects, the **somGetClass** method should normally be used to access the class of an object.

In a few special cases, it is not possible to make a method call on an object in order to determine its class. For such situations, SOM provides the **SOM_GetClass** macro. In general, the **somGetClass** method and the **SOM_GetClass** macro may have different behavior (if **somGetClass** has been overridden). This difference may be limited to side effects, but it is possible for their results to differ as well. The **SOM_GetClass** macro should only be used when absolutely necessary.

Creating a class object

A class object is created automatically the first time the <*className*>**New** macro (for C) or the **new** operator (C++) is invoked to create an instance of that class. In other situations, however, it may be necessary to create a class object explicitly, as this section describes.

Using <*className*>Renew or somRenew

It is sometimes necessary to create a class object before creating any instances of the class. For example, creating instances using the <*className*>**Renew** macro or the **somRenew** method requires knowing how large the created instance will be, so that memory can be allocated for it. Getting this information requires creating the class object (see the example under “Creating instances of a class” early in this chapter). As another example, a class object must be explicitly created when a program does not use the SOM bindings for a class. Without SOM bindings for a class, its instances must be created using **somNew** or **somRenew**, and these methods require that the class object be created in advance.

Use the <*className*>**NewClass** procedure to create a class object :

- When using the C/C++ language bindings for the class, and
- When the name of the class is known at compile time.

Using <*className*>NewClass

The <*className*>**NewClass** procedure initializes the SOM run-time environment, if necessary, creates the class object (unless it already exists), creates class objects for the ancestor classes and metaclass of the class, if necessary, and returns a pointer to the newly created class object. After its creation, the class object can be referenced in client code using the macro

_<*className*> (for C and C++ programs)

or the expression

<*className*>**ClassData.classObject** (for C and C++ programs).

The <*className*>**NewClass** procedure takes two arguments, the major version number and minor version number of the class. These numbers are checked against the version numbers built into the class library to determine if the class is compatible with the client's expectations. The class is compatible if it has the same major version number and the same or a higher minor version number. If the class is not compatible, an error is raised. Major version numbers usually only change when a significant enhancement or incompatible change is made to a class. Minor version numbers change when minor enhancements or fixes are made. Downward compatibility is usually maintained across changes in the minor version number. Zero can be used in place of version numbers to bypass version number checking.

When using SOM bindings for a class, these bindings define constants representing the major and minor version numbers of the class at the time the bindings were generated. These constants are named <*className*>_MajorVersion and <*className*>_MinorVersion. For example, the following procedure call:

```
AnimalNewClass(Animal_MajorVersion, Animal_MinorVersion);
```

creates the class object for class "Animal". Thereafter, `_Animal` can be used to reference the "Animal" class object.

The preceding technique for checking version numbers is not failsafe. For performance reasons, the version numbers for a class are only checked when the class object is created, and not when the class object or its instances are used. Thus, run-time errors may result when usage bindings for a particular version of a class are used to invoke methods on objects created by an earlier version of the class.

Using somFindClass or somFindClsInFile

To create a class object when *not* using the C/C++ language bindings for the class, or when the class name is *not* known at compile time:

- First, initialize the SOM run-time environment by calling the **somEnvironmentNew** function (unless it is known that the SOM run-time environment has already been initialized).
- Then, use the **somFindClass** or **somFindClsInFile** method to create the class object. (The class must already be defined in a dynamically linked library, or DLL.)

The **somEnvironmentNew** function initializes the SOM run-time environment. That is, it creates the four primitive SOM objects (**SOMClass, SOMObject, SOMClassMgr,** and the **SOMClassMgrObject**), and it initializes SOM global variables. The function takes no arguments and returns a pointer to the **SOMClassMgrObject**.

Note: Although **somEnvironmentNew** must be called before using other SOM functions and methods, explicitly calling **somEnvironmentNew** is usually not necessary when using the C/C++ bindings, because the macros for <*className*>**NewClass**, <*className*>**New**, and <*className*>**Renew** call it automatically, as does the **new** operator for C++. Calling **somEnvironmentNew** repeatedly does no harm.

After the SOM run-time environment has been initialized, the methods **somFindClass** and **somFindClsInFile** can be used to create a class object. These methods must be invoked on the class manager, which is pointed to by the global variable **SOMClassMgrObject.** (It is also returned as the result of **somEnvironmentNew**.)

The **somFindClass** method takes the following arguments:

classId	— A **somId** identifying the name of the class to be created. The **somIdFromString** function returns a *classId* given the name of the class.
major version number	— The expected major version number of the class.
minor version number	— The expected minor version number of the class.

The version numbers are checked against the version numbers built into the class library to determine if the class is compatible with the client's expectations.

The **somFindClass** method dynamically loads the DLL containing the class's implementation, if needed, creates the class object (unless it already exists) by invoking its <*className*>**NewClass** procedure, and returns a pointer to it. If the class could not be created, **somFindClass** returns NULL. For example, the following C code fragment creates the class "Hello" and stores a pointer to it in "myClass":

```
SOMClassMgr cm = somEnvironmentNew();
somId classId = somIdFromString("Hello");
SOMClass myClass = _somFindClass(SOMClassMgrObject, classId,
                      Hello_MajorVersion, Hello_MinorVersion);
. . .
SOMFree(classId);
```

The **somFindClass** method uses **somLocateClassFile** to get the name of the library file containing the class. If the class was defined with a "dllname" class modifier, then **somLocateClassFile** returns that file name; otherwise, it assumes that the class name is the name of the library file. The **somFindClsInFile** method is similar to **somFindClass,** except that it takes an additional (final) argument — the name of the library file containing the class. The **somFindClsInFile** method is useful when a class is packaged in a DLL along with other classes and the "dllname" class modifier has not been given in the class's IDL specification.

Warning: On AIX and Windows, the **somFindClass** and **somFindClsInFile** methods should *not* be used to create a class whose implementation is statically linked with the client program. Instead, the class object should be created using the <*className*>**NewClass** procedure provided by the class's .h/.xh header file. Static linkage is not created by simply including usage bindings in a program, but by use of the offset-resolution method-invocation macros.

Referring to class objects

Saving a pointer as the class object is created: The <*className*>**NewClass** macro and the **somFindClass** method, used to create class objects, both return a pointer to the newly created class object. Hence, one way to obtain a pointer to a class object is to save the value returned by <*className*>**NewClass** or **somFindClass** when the class object is created.

Getting a pointer after the class object is created: After a class object has been created, client programs can also get a pointer to the class object from the class name. When the class name is known at compile time and the client program is using the C or C++ language bindings, the macro

_<*className*>

can be used to refer to the class object for <*className*>. Also, when the class name is known at compile time and the client program is using the C or C++ language bindings, the expression

<*className*>**ClassData.classObject**

refers to the class object for <*className*>. For example, `_Hello` refers to the class object for class “Hello” in C or C++ programs, and `HelloClassData.classObject` refers to the class object for class “Hello.” in C or C++ programs.

Getting a pointer to the class object from an instance: If any instances of the class are known to exist, a pointer to the class object can also be obtained by invoking the **somGetClass** method on such an instance. (See “Getting the class of an object,” above.)

Getting a pointer in other situations: If the class name is *not* known until run time, or if the client program is *not* using the C or C++ language bindings, and *no* instances of the class are known to exist, then the **somClassFromId** method can be used to obtain a pointer to a class object after the class object has been created. The **somClassFromId** method should be invoked on the class manager, which is pointed to by the global variable **SOMClassMgrObject.** The only argument to the method is a **somId** for the class name, which can be obtained using the **somIdFromString** function. The method **somClassFromId** returns a pointer to the class object of the specified class. For example, the following C code stores in “myClass” a pointer to the class object for class “Hello” (or NULL, if the class cannot be located):

```
SOMClassMgr cm   =  somEnvironmentNew();
somId classId    =  somIdFromString("Hello");
SOMClass myClass = _somClassFromId(SOMClassMgrObject, classId,
                        Hello_MajorVersion, Hello_MinorVersion);
SOMFree(classId);
```

Compiling and linking

This section describes how to compile and link C and C++ client programs. Compiling and linking a client program with a SOM class is done in one of two ways, depending upon whether or not the class is packaged as a library, as described below.

Note: If you are building an application that uses a combination of C and C++ compiled object modules, then the C++ linker must be used to link them.

If the class is *not* packaged as a library (that is, the client program has the implementation source code for the class, as in the examples given in the SOM IDL tutorial), then the client program can be compiled together with the class implementation file as follows. (This assumes that the client program and the class are both implemented in the same language, C or C++. If this is not the case, then each module must be compiled separately to produce an object file and the resulting object files linked together to form an executable.)

In the following examples, the environment variable SOMBASE refers to the directory in which SOM has been installed. The examples also assume that the header files and the import library for the “Hello” class reside in the “include” and “lib” directories where SOM has been installed. If this is not the case, additional path information should be supplied for these files. For client program “main” and class “Hello”:

Under AIX, for **C** programmers:

```
> xlc -I. -I$SOMBASE/include main.c hello.c \
      -L$SOMBASE/lib -lsomtk -o main
```

Under AIX, for **C++** programmers:

```
> xlC -I. -I$SOMBASE/include main.C hello.C \
      -L$SOMBASE/lib -lsomtk -o main
```

Under OS/2, for **C** programmers:

```
> set LIB=%SOMBASE%\lib;%LIB%
> icc -I. -I%SOMBASE%\include main.c hello.c somtk.lib
```

Under OS/2, for **C++** programmers:

```
> set LIB=%SOMBASE%\lib;%LIB%
> icc -I. -I%SOMBASE%\include main.cpp hello.cpp somtk.lib
```

Under Windows, for **C** programmers:

```
> cl -AL -Zp1 -I. -I%SOMBASE%\include \
      main.c hello.c llibcew.lib libw.lib somtk.lib main.def
```

Under Windows, for **C++** programmers:

```
> cl -AL -Zp1 -I. -I%SOMBASE%\include \
      main.cpp hello.cpp llibcew.lib libw.lib somtk.lib main.def
```

Windows compiler notes: The SOM Compiler and runtime libraries under Windows only provide support for Large Memory Model systems. The "–Oi" compile flag should be used whenever a SOM Windows program contains or calls any functions that return floats or doubles. The "–F" option can be used to set the stack size if STACKSIZE is not specified in a .def file. Additionally, for this manual, compile and link commands are demonstrated using the Visual C++ compiler. Sample Makefiles shipped with SOMobjects For Windows also show how to build with other compilers.

If the class is packaged as a class library, then the client program, "main", is compiled as above, except that the class implementation file is not part of the compilation. Instead, the "import library" provided with the class library is used to resolve the symbolic references that appear in "main". For example, to compile the C client program "main.c" that uses class "Hello":

Under AIX:

```
> xlc -I. -I$SOMBASE/include main.c -lc -L$SOMBASE/lib -lsomtk \
      -lhello -o main
```

Under OS/2:

```
> set LIB=%SOMBASE%\lib;%LIB%
> icc -I. -I%SOMBASE%\include main.c somtk.lib hello.lib
```

Under Windows:

```
> cl -AL -Zp1 -I. -I%SOMBASE%\include \
      main.c llibcew.lib libw.lib somtk.lib hello.lib main.def
```

Windows users: See also "Windows Compiler Notes" above.

3.3 Language-neutral Methods and Functions

This section describes methods, functions, and macros that client programs can use regardless of the programming language in which they are written. In other words, these functions and methods are *not* part of the C or C++ bindings.

Generating output

The following functions and methods are used to generate output, including descriptions of SOM objects. They all produce their output using the character-output procedure held by the global variable **SOMOutCharRoutine**. The default procedure for character output simply writes the character to *stdout*, but it can be replaced to change the output destination of the methods and functions below. (See Chapter 5 for more information on customizing SOM.)

somDumpSelf — (method) Writes a detailed description of an object, including its class, its location, and its instance data. The receiver of the method is the object to be dumped. An additional argument is the "nesting level" for the description. [All lines in the description will be indented by (2 * level) spaces.]

somPrintSelf — (method) Writes a brief description of an object, including its class and location in memory. The receiver of the method is the object to be printed.

somPrintf — (function) SOM's version of the C "printf" function. It generates character stream output via **SOMOutCharRoutine**. It has the same interface as the C "printf" function.

somVprintf — (function) Represents the "vprint" form of **somPrintf**. Its arguments are a formatting string and a **va_list** holding the remaining arguments.

somPrefixLevel — (function) Generates (via **somPrintf**) spaces to prefix a line at the indicated level. The return type is **void**. The argument is an integer specifying the level. The number of spaces generated is (2 * level).

somLPrintf — (function) Combines **somPrefixLevel** and **somPrintf.** The first argument is the level of the description (as for **somPrefixLevel**) and the remaining arguments are as for **somPrintf** (or for the C "printf" function).

See the *SOMobjects Developer Toolkit Programmers Reference Manual* for more information on a specific function or method.

Getting information about a class

The following methods are used to obtain information about a class or to locate a particular class object:

somCheckVersion — Checks a class for compatibility with the specified major and minor version numbers. The receiver of the method is the SOM class about which information is needed. Additional arguments are values of the major and minor version numbers. The method returns TRUE if the class is compatible, or FALSE otherwise.

somClassFromId — Finds the class object of an existing class when given its **somId**, but without loading the class. The receiver of the method is the class manager (pointed to by the global variable **SOMClassMgrObject**). The additional argument is the class's **somId**. The method returns a pointer to the class (or NULL if the class does not exist).

somDescendedFrom — Tests whether one class is derived from another. The receiver of the method is the class to be tested, and the potential ancestor class is the argument. The method returns TRUE if the relationships exists, or FALSE otherwise.

somFindClass — Finds or creates the class object for a class, given the class's **somId** and its major and minor version numbers. The receiver of the method is the class manager (pointed to by the global variable **SOMClassMgrObject**). Additional arguments are the class's **somId** and the major and minor version numbers. The method returns a pointer to the class object, or NULL if the class could not be created.

somFindClsInFile — Finds or creates the class object for a class. This method is similar to **somFindClass**, except the user also provides the name of a file to be used for dynamic loading, if needed. The receiver of the method is the class manager (pointed to by the global variable **SOMClassMgrObject**). Additional arguments are the class's **somId**, the major and minor version numbers, and the file name. The method returns a pointer to the class object, or NULL if the class could not be created.

somGetInstancePartSize — Obtains the size of the instance variables introduced by a class. The receiver of the method is the class object. The method returns the amount of space, in bytes, needed for the instance variables.

somGetInstanceSize — Obtains the total size requirements for an instance of a class. The receiver of the method is the class object. The method returns the amount of space, in bytes, required for the instance variables introduced by the class itself and by all of its ancestor classes.

somGetName — Obtains the name of a class. The receiver of the method is the class object. The method returns the class name.

somGetNumMethods — Obtains the number of methods available for a class. The receiver of the method is the class object. The method returns the total number of currently available methods (static or otherwise, including inherited methods).

somGetNumStaticMethods — Obtains the number of static methods available for a class. (A static method is one declared in the class's interface specification [.idl] file.) The receiver of the method is the class object. The method returns the total number of available static methods, including inherited ones.

somGetParents — Obtains a sequence of the parent (base) classes of a specified class. The receiver of the method is the class object. The method returns a pointer to a linked list of the parent (base) classes (unless the receiver is **SOMObject**, for which it returns NULL).

somGetVersionNumbers — Obtains the major and minor version numbers of a class. The receiver of the method is the class object. The return type is void, and the two arguments are pointers to locations in memory where the method can store the major and minor version numbers (of type **long**).

somSupportsMethod — Indicates whether instances of a given class support a given method. The receiver of the **somSupportsMethod** method is the class object. The argument is the **somId** for the method in question. The **somSupportsMethod** method returns TRUE if the method is supported, or FALSE otherwise.

See the *SOMobjects Developer Toolkit Programmers Reference Manual* for more information on a specific method.

Getting information about an object

The following methods and functions are used to obtain information about an object (instance) or to determine whether a variable holds a valid SOM object.

Methods

somGetClass — Gets the class object of a specified object. The receiver of the method is the object whose class is desired. The method returns a pointer to the object's corresponding class object.

somGetClassName — Obtains the class name of an object. The receiver of the method is the object whose class name is desired. The method returns a pointer to the name of the class of which the specified object is an instance.

somGetSize — Obtains the size of an object. The receiver of the method is the object. The method returns the amount of contiguous space, in bytes, that is needed to hold the object itself (not including any additional space that the object may be using or managing outside of this area).

somIsA — Determines whether an object is an instance of a given class or of one of its descendant classes. The receiver of the method is the object to be tested. An additional argument is the name of the class to which the object will be compared. This method returns TRUE if the object is an instance of the specified class or if (unlike **somIsInstanceOf**) it is an instance of any descendant class of the given class; otherwise, the method returns FALSE.

somIsInstanceOf — Determines whether an object is an instance of a specific class (but not of any descendant class).The receiver of the method is the object. The argument is the name of the class to which the object will be compared. The method returns TRUE if the object is an instance of the specified class, or FALSE otherwise.

somRespondsTo — Determines whether an object supports a given method. The receiver of the method is the object. The argument is the **somId** for the method in question. (A **somId** can be obtained from a string by using the **somIdFromString** function.) The **somRespondsTo** method returns TRUE if the object supports the method, or FALSE otherwise.

Functions

somIsObj — Takes as its only argument an address (which may not be valid). The function returns TRUE (1) if the address contains a valid SOM object, or FALSE (0) otherwise. This function is designed to be failsafe.

See the *SOMobjects Developer Toolkit Programmers Reference Manual* for more information on a specific method or function.

Debugging

The following macros are used to conditionally generate output for debugging. All output generated by these macros is written using the replaceable character-output procedure pointed to by the global variable **SOMOutCharRoutine**. The default procedure simply writes the character to *stdout*, but it can be replaced to change the output destination of the methods and functions below. (See Chapter 5, "Implementing Classes in SOM," for more information on customizing SOM.)

Debugging output is produced or suppressed based on the settings of three global variables, **SOM_TraceLevel, SOM_WarnLevel,** and **SOM_AssertLevel:**

- **SOM_TraceLevel** controls the behavior of the <*className*>**MethodDebug** macro;
- **SOM_WarnLevel** controls the behavior of the macros **SOM_WarnMsg**, **SOM_TestC,** and **SOM_Expect;** and
- **SOM_AssertLevel** controls the behavior of the **SOM_Assert** macro.

Available macros for generating debugging output are as follows:

<*className*>**MethodDebug**

— (macro for C and C++ programmers using the SOM language bindings for <*className*>)
The arguments to this macro are a class name and a method name. If the **SOM_TraceLevel** global variable has a nonzero value, the <className>**MethodDebug** macro produces a message each time the specified method (as defined by the specified class) is executed. This macro is typically used within the procedure that implements the specified method. (The SOM Compiler automatically generates calls to the <*className*>**MethodDebug** macro within the implementation template files it produces.) To suppress method tracing for all methods of a class, put the following statement in the implementation file after including the header file for the class:

```
#define <className>MethodDebug(c,m) \
                        SOM_NoTrace(c,m)
```

This can yield a slight performance improvement. The **SOMMTraced** metaclass, discussed below, provides a more extensive tracing facility that includes method parameters and returned values.

SOM_TestC — The **SOM_TestC** macro takes as an argument a boolean expression. If the boolean expression is TRUE (nonzero) and **SOM_AssertLevel** is greater than zero, then an informational message is output. If the expression is FALSE (zero) and **SOM_WarnLevel** is greater than zero, a warning message is produced.

SOM_WarnMsg — The **SOM_WarnMsg** macro takes as an argument a character string. If the value of **SOM_WarnLevel** is greater than zero, the specified message is output.

SOM_Assert — The **SOM_Assert** macro takes as arguments a boolean expression and an error code (an integer). If the boolean expression is TRUE (nonzero) and **SOM_AssertLevel** is greater than zero, then an informational message is output. If the expression is FALSE (zero), and the error code indicates a warning-level error and **SOM_WarnLevel** is greater than zero, then a warning message is output. If the expression is FALSE and the error code indicates a fatal error, then an error message is produced and the process is terminated.

SOM_Expect — The **SOM_Expect** macro takes as an argument a boolean expression. If the boolean expression is FALSE (zero) and **SOM_WarnLevel** is set to be greater than zero, then a warning message is output. If *condition* is TRUE and **SOM_AssertLevel** is set to be greater than zero, then an informational message is output.

See the *SOMobjects Developer Toolkit Programmers Reference Manual* for more information on a specific macro.

The **somDumpSelf** and **somPrintSelf** methods can be useful in testing and debugging. The **somPrintSelf** method produces a brief description of an object, and the **somDumpSelf** method produces a more detailed description. See the *SOMobjects Developer Toolkit Programmers Reference Manual* for more information.

Checking the validity of method calls

The C and C++ language bindings include code to check the validity of method calls at run time. If a validity check fails, the **SOM_Error** macro ends the process. (**SOM_Error** is described below.) To enable method-call validity checking, place the following directive in the client program prior to any *#include* directives for SOM header files:

```
#define SOM_TestOn
```

Alternatively, the **–DSOM_TestOn** option can be used when compiling the client program to enable method-call validity checking.

Exceptions and error handling

In the classes provided in the SOM run-time library (that is, **SOMClass, SOMObject,** and **SOMClassMgr**), error handling is performed by a user-replaceable procedure, pointed to by the global variable **SOMError**, that produces an error message and an error code and, if appropriate, ends the process where the error occurred. (Chapter 5 describes how to customize the error handling procedure.)

Each error is assigned a unique integer error code. Errors are grouped into three categories, based on the last digit of the error code:

SOM_Ignore — This category of error represents an informational event. The event is considered normal and can be ignored or logged at the user's discretion. Error codes having a last digit of 2 belong to this category.

SOM_Warn — This category of error represents an unusual condition that is not a normal event, but is not severe enough to require program termination. Error codes having a last digit of 1 belong to this category.

SOM_Fatal — This category of error represents a condition that should not occur or that would result in loss of system integrity if processing were allowed to continue. In the default error handling procedure, these errors cause the termination of the process in which they occur. Error codes having a last digit of 9 belong to this category.

The various codes for all errors detected by SOM are listed in Appendix A, "Customer Support and Error Codes."

When errors are encountered in client programs or user defined-classes, the following two macros can be used to invoke the error-handling procedure:

SOM_Error — The **SOM_Error** macro takes an error code as its only argument and invokes the SOM error handling procedure (pointed to by the global variable **SOMError**) to handle the error. The default error handling procedure prints a message that includes the error code, the name of the source file, and the line number where the macro was invoked. If the last digit of the error code indicates a serious error (of category **SOM_Fatal**), the process causing the error is terminated. (Chapter 5 describes how to customize the error handling procedure.)

SOM_Test — The **SOM_Test** macro takes a boolean expression as an argument. If the expression is TRUE (nonzero) and the **SOM_AssertLevel** is greater than zero, then an informational message is output. If the expression is FALSE (zero), an error message is produced and the program is terminated.

See the *SOMobjects Developer Toolkit Programmers Reference Manual* for more information on a specific macro.

Other classes provided by the SOMobjects Toolkit (including those in the Persistence, Replication, DSOM, and Interface Repository frameworks, and the utility classes and metaclasses) handle errors differently. Rather than invoking **SOMError** with an error code, their methods return *exceptions* via the (**Environment ***) inout parameter required by these methods. The following sections describe the exception declarations, the standard exceptions, and how to set and get exception information in an **Environment** structure.

Exception declarations

As discussed in Chapter 4 in Section 4.2 entitled "SOM Interface Definition Language," a method may be declared to return zero or more **exceptions**. IDL exceptions are implemented by simply passing back error information after a method call, as opposed to the "catch/throw" model where an exception is implemented by a long jump or signal. Associated with each type of exception is a name and, optionally, a struct-like data structure for holding error information. A method declares the types of exceptions it may return in a **raises** expression.

Below is an example IDL declaration of a "BAD_FLAG" exception, which may be "raised" by a "checkFlag" method, as part of a "MyObject" interface:

```
interface MyObject {
   exception BAD_FLAG { long ErrCode; char Reason[80]; };

   void checkFlag(in unsigned long flag) raises(BAD_FLAG);
};
```

An exception structure contains whatever information is necessary to help the caller understand the nature of the error. The exception declaration can be treated like a **struct** definition: that is, whatever you can access in an IDL **struct**, you can access in an **exception** declaration. Alternatively, the structure can be *empty*, whereby the exception is just identified by its name.

The SOM Compiler will map the exception declaration in the above example to the following C language constructs:

```
typedef struct BAD_FLAG {
   long ErrCode;
   char Reason[80];
} BAD_FLAG;

#define ex_BAD_FLAG "MyObject::BAD_FLAG"
```

When an exception is detected, the "checkFlag" method must call **SOMMalloc** to allocate a "BAD_FLAG" structure, initialize it with the appropriate error information, and make a call to **somSetException** (see "Setting an exception value," below) to record the exception value in the **Environment** structure passed in the method call. The caller, after invoking "checkFlag", can check the **Environment** structure that was passed to the method to see if there was an exception, and if so, extract the exception value from the **Environment** (see "Getting an exception value," below.)

Standard exceptions

In addition to user-defined exceptions (those defined explicitly in an IDL file), there are several predefined exceptions for system run-time errors. A *system exception* can be returned on any method call. (That is, they are implicitly declared for every method whose class uses IDL call style, and they do not appear in any **raises** expressions.) The standard exceptions are listed in Table 2 of Section 4.2, "SOM Interface Definition Language" in Chapter 4, "SOM IDL and the SOM Compiler." Most of the predefined system exceptions pertain to Object Request Broker errors. Consequently, these types of exceptions are most likely to occur in DSOM applications (Chapter 6).

Each of the standard exceptions has the same structure: an error code (to designate the subcategory of the exception) and a completion status code. For example, the NO_MEMORY standard exception has the following definition:

```
enum completion_status {YES, NO, MAYBE};
exception NO_MEMORY { unsigned long minor;
                      completion_status completed; };
```

The completion status value indicates whether the method was never initiated (NO), completed execution prior to the exception (YES), or the completion status is indeterminate (MAYBE).

Since all the standard exceptions have the same structure, file "somcorba.h" (included by "som.h") defines a generic **StExcep** typedef which can be used instead of the specific typedefs:

```
typedef struct StExcep {
     unsigned long minor;
     completion_status completed;
} StExcep;
```

The standard exceptions are defined in an IDL module called **StExcep**, in the file named "stexcep.idl", and the C definitions can be found in "stexcep.h".

The Environment

The **Environment** is a data structure that contains environmental information that can be passed between a caller and a called object when a method is executed. For example, it is used to pass information about the user id of a client, to return exception data to the client following a method call, and so on.

A pointer to an **Environment** variable is passed as an argument to method calls (unless the method's class has the **callstyle=oidl** SOM IDL modifier). The **Environment** typedef is defined in "som.h", and an instance of the structure is allocated by the caller in any reasonable way: on the stack (by declaring a local variable and initializing it using the macro **SOM_InitEnvironment**), dynamically (using the **SOM_CreateLocalEnvironment** macro), or by calling the **somGetGlobalEnvironment** function to allocate an **Environment** structure to be shared by objects running in the same thread.

For class libraries that use **callstyle=oidl**, there is no explicit **Environment** parameter. For these libraries, exception information may be passed using the per-thread **Environment** structure returned by the **somGetGlobalEnvironment** procedure.

Setting an exception value

To set an exception value in the caller's **Environment** structure, a method implementation makes a call to the **somSetException** procedure:

```
void  somSetException ( Environment *ev,
                        exception_type major,
                        string exception_name,
                        void *params);
```

where "ev" is a pointer to the **Environment** structure passed to the method, "major" is an **exception_type,** "exception_name" is the string name of the exception (usually the constant defined by the IDL compiler, for example, `ex_BAD_FLAG`), and "params" is a pointer to an (initialized) exception structure which must be allocated by **SOMMalloc:**

```
typedef enum exception_type {
   NO_EXCEPTION, USER_EXCEPTION, SYSTEM_EXCEPTION
} exception_type;
```

It is important to reiterate that **somSetException** expects the *params* argument to be a pointer to a structure that was allocated using **SOMMalloc**. When **somSetException** is called, the client *passes ownership* of the exception structure to the SOM run-time environment. The SOM run-time environment will free the structure when the exception is reset (that is, upon the next call to **somSetException**), or when the **somExceptionFree** function is called.

Note that **somSetException** simply sets the exception value; it performs no exit processing. If there are multiple calls to **somSetException** before the method returns, the caller will only see the last exception value.

Getting an exception value

After a method returns, the calling client program can look at the **Environment** structure to see if there was an exception. The **Environment** struct is mostly opaque, except for an exception type field named **_major**:

```
typedef struct Environment {
   exception_type   _major;
   ...
} Environment;
```

If **ev._major != NO_EXCEPTION**, there was an exception returned by the call. The caller can retrieve the exception name and value (passed as parameters in the **somSetException** call) from an **Environment** struct via the following functions:

```
string  somExceptionId (Environment *ev);
somToken  somExceptionValue (Environment *ev);
```

The **somExceptionId** function returns the exception name, if any, as a string. The function **somExceptionValue** returns a pointer to the value of the exception, if any, contained in the exception structure. If NULL is passed as the **Environment** pointer in either of the above calls, an implicit call is made to **somGetGlobalEnvironment**.

The **somExceptionFree** function will free any memory in the **Environment** associated with the last exception:

void somExceptionFree (Environment ***ev*****);**

If preferred, developers can alternatively use the CORBA "exception_free" API to free the memory in an **Environment** structure.

Note**:** File "somcorba.h" (included by "som.h") provides the following aliases for strict compliance with CORBA programming interfaces:

```
#ifdef CORBA_FUNCTION_NAMES
#define exception_id    somExceptionId
#define exception_value somExceptionValue
#define exception_free  somExceptionFree
#endif /* CORBA_FUNCTION_NAMES */
```

Example

The following IDL interface for a "MyObject" object (in a file called "myobject.idl") declares a "BAD_FLAG" exception, which can be raised by the "checkFlag" method,:

```
interface MyObject {
   exception BAD_FLAG { long ErrCode; char Reason[80]; };

   void checkFlag(in unsigned long flag) raises(BAD_FLAG);
};
```

The SOM IDL compiler will map the exception to the following C language constructs, in "myobject.h"**:**

```
typedef struct BAD_FLAG {
   long ErrCode;
   char Reason[80];
} BAD_FLAG;

#define ex_BAD_FLAG "MyObject::BAD_FLAG"
```

A client program that invokes the "checkFlag" method might contain the following error handling code. (Note: The error checking code below lies in the user-written procedure, "ErrorCheck," so the code need not be replicated through the program.)

```
#include ”som.h”
#include ”myobject.h”

boolean ErrorCheck(Environment *ev);   /* prototype */

main()
{
   unsigned long flag;
   Environment ev;
   MyObject myobj;
   char     *exId;
   BAD_FLAG *badFlag;
   StExcep  *stExValue;

   myobj = MyObjectNew();
   flag  = 0x01L;
   SOM_InitEnvironment(&ev);

    /* invoke the checkFlag method, passing the Environment param */
   _checkFlag(myobj, &ev, flag);

   /* check for exception */
   if (ErrorCheck(&ev))
   {
      /* ... */
      somExceptionFree(&ev);   /* free the exception memory */
   }

   /* ... */
}


/* error checking procedure */

boolean ErrorCheck(Environment *ev)
{
   switch (ev._major)
   {
   case SYSTEM_EXCEPTION:
      /* get system exception id and value */
      exId      = somExceptionId(ev);
      stExValue = somExceptionValue(ev);
      /* ... */
      return(TRUE);

   case USER_EXCEPTION:
      /* get user-defined exception id and value */
      exId = somExceptionId(ev);
      if (strcmp(exId, ex_BAD_FLAG) == 0)
      {
         badFlag = (BAD_FLAG *) somExceptionValue(ev);
         /* ... */
      }
      /* ... */
      return(TRUE);

   case NO_EXCEPTION:
      return(FALSE);
   }
}
```

The implementation of the “checkFlag” method might contain the following error-handling code:

```
#include ”som.h”
#include ”myobject.h”

void checkFlag(MyObject somSelf, Environment *ev,
               unsigned long flag)
{
   BAD_FLAG *badFlag;
   /* ... */

   if ( /* flag is invalid */ )
   {
      badFlag = (BAD_FLAG *) SOMMalloc(sizeof(BAD_FLAG));
      badFlag->ErrCode = /* bad flag code */;
      strcpy(badFlag->Reason, ”bad flag was passed”);
      somSetException(ev, USER_EXCEPTION,
                      ex_BAD_FLAG, (void *)badFlag);
      return;
   }
   /* ... */
}
```

Memory management

The SOMobjects Toolkit provides several functions that can be used for memory management.

Using SOM equivalents to ANSI C functions

The memory management functions used by SOM are a subset of those supplied in the ANSI C standard library. They have the same calling interface and the same return types as their ANSI C equivalents, but include supplemental error checking. Errors detected by these functions are passed to **SOMError** (described in the previous section). The correspondence between SOM memory management functions and their ANSI C standard library equivalents is shown below:

SOM Function	Equivalent ANSI C Library Routine
SOMMalloc	**malloc**
SOMCalloc	**calloc**
SOMRealloc	**realloc**
SOMFree	**free**

SOMMalloc, SOMCalloc, SOMRealloc, and **SOMFree** are actually *global variables* that point to the SOM memory management functions (rather than being the names of the functions themselves), so that users can replace them with their own memory management functions if desired. (See Chapter 5 for information on replacing the SOM memory management functions.)

Clearing memory for objects

The memory associated with objects initialized by a client program must also be freed by the client. The SOM-provided method **somFree** is used to release the storage containing the receiver object:

```
#include ”origcls.h”

main ()
{
   OrigCls myObject;
   myObject = OrigClsNew ();

    /*  Code to use myObject  */

   _somFree (myObject);
}
```

Clearing memory for the Environment

Any memory associated with an exception in an **Environment** structure is typically freed using the **somExceptionFree** function. (Or, the CORBA "exception_free" API can be used.) The **somExceptionFree** function takes the following form (also see "Example" in the previous topic for an application example):

void somExceptionFree (Environment ***ev*****);**

Note: For information on managing the memory, objects, and exceptions used by DSOM applications, see "Memory management for DSOM" in Chapter 6, "Distributed SOM (DSOM)."

SOM manipulations using somId's

A **somId** is similar to a number that represents a zero-terminated **string**. A **somId** is used in SOM to identify method names, class names, and so forth. For example, many of the SOM methods that take a method or class name as a parameter require a value of type **somId** rather than **string**. All SOM manipulations using **somId**s are case insensitive, although the original case of the **string** is preserved.

During its first use with any of the following functions, a **somId** is automatically converted to an internal representation (registered). Because the representation of a **somId** changes, a special SOM type (**somId**) is provided for this purpose. Names and the corresponding **somId** can be declared at compile time, as follows:

```
string example = "exampleMethodName";
somId exampleId = &example;
```

or a **somId** can be generated at run time, as follows:

```
somId myMethodId;
myMethodId = somIdFromString("exampleMethodName");
```

SOM provides the following functions that generate or use a **somId**:

somIdFromString — Finds the **somId** that corresponds to a **string**. The method takes a **string** as its argument, and returns a value of type **somId** that represents the string. The returned **somId** must later be freed using **SOMFree.**

somStringFromId — Obtains the **string** that corresponds to a **somId**. The function takes a **somId** as its argument and returns the **string** that the **somId** represents.

somCompareIds — Determines whether two **somId** values are the same (that is, represent the same **string**). This function takes two **somId** values as arguments. It returns TRUE (1) if the **somId**s represent the same **string**, or FALSE (0) otherwise.

somCheckId — Determines whether SOM already knows a **somId**. The function takes a **somId** as its argument. It verifies whether the **somId** is registered and in normal form, registers it if necessary, and returns the input **somId**.

somRegisterId — The same as **somCheckId,** except it returns TRUE (1) if this is the first time the **somId** has been registered, or FALSE (0) otherwise.

somUniqueKey — Finds the unique key for a **somId.** The function takes a **somId** identifier as its argument, and returns the unique key for the **somId** — a number that uniquely represents the **string** that the **somId** represents. This key is the same as the key for another **somId** if and only if the other **somId** refers to the same **string** as the input **somId**.

somTotalRegIds — Finds the total number of **somId**s that have been registered, as an **unsigned long**. This function is used to determine an appropriate argument to **somSetExpectedIds**, below, in later executions of the program. The function takes no input arguments.

somSetExpectedIds — Indicates how many unique **somId**s SOM can expect to use during program execution, which, if accurate, can improve the space and time utilization of the program slightly. This routine must be called before the SOM run-time environment is initialized (that is, before the function **somEnvironmentNew** is invoked and before any objects are created). This is the only SOM function that can be invoked before the SOM run-time environment is initialized. The input argument is an **unsigned long**. The function has no return value.

somBeginPersistentIds and **somEndPersistentIds** — Delimit a time interval for the current thread during which it is guaranteed that (a) any new **somId**s that are created will refer only to static strings and (b) these strings will not be subsequently modified or freed. These functions are useful because **somId**s that are registered within a "persistent ID interval" can be handled more efficiently.

See the *SOMobjects Developer Toolkit Programmers Reference Manual* for more information on a specific function.