VisualAge for C++ for Windows

S33H-5044-00

SOM Programming Reference

Version 3.5

VisualAge for C++ for Windows

S33H-5044-00

SOM Programming Reference

Version 3.5

Note!

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

First Edition (February 1996)

This edition applies to Version 3.5 of IBM VisualAge for C++ for Windows (33H4979, 33H4980) and to all subsequent releases and modifications until otherwise indicated in new editions. Make sure you are using the correct edition for the level of the product.

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

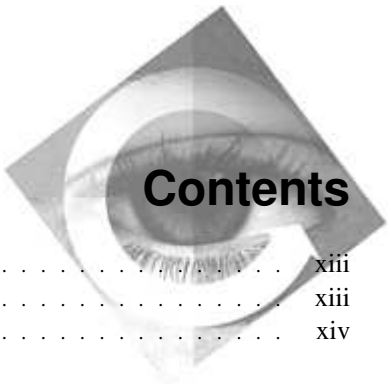
A form for readers' comments is provided at the back of this publication. If the form has been removed, address your comments to:

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

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

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

© Copyright International Business Machines Corporation 1994, 1996. All rights reserved.
Note to U.S. Government Users — Documentation related to restricted rights — Use, duplication or disclosure is subject to restrictions set forth in GSA ADP Schedule Contract with IBM Corp.



Contents

Notices xiii
Programming Interface Information xiii
Trademarks and Service Marks xiv

About This Book xv
How This Book Is Organized xv
Who Should Use This Book xvii
How to Get Help xvii
Getting Help Inside VisualAge for C++ xviii
Getting Help from the Command Line xviii
Getting Help for a Keyword or Construct xix
Online Documents Available in VisualAge for C++ xix

Chapter 1. SOM Kernel Reference 1
somApply 3
somBuildClass 6
somCheckId 8
somClassResolve 9
somCompareIds 12
somDataResolve 14
somDataResolveChk 16
somEnvironmentEnd 18
somEnvironmentNew 19
somExceptionFree 21
somExceptionId 23
somExceptionValue 25
somGetGlobalEnvironment 27
somIdFromString 29
somIsObj 30
somLPrintf 32
somMainProgram 34
somParentNumResolve 35
somParentResolve 38
somPrefixLevel 40
somPrintf 41
somRegisterId 43
somResolve 45
somResolveByName 48
somSetException 50
somSetExpectedIds 53

© Copyright IBM Corp. 1994, 1996

somSetOutChar 55
somStringFromId 56
somTotalRegIds 57
somUniqueKey 59
somVaBuf_add 61
somVaBuf_create 63
somVaBuf_destroy 66
somVaBuf_get_valist 67
somvalistGetTarget 68
somvalistSetTarget 70
somVprintf 72
SOMCalloc 74
somClassInitFuncName 76
SOMDeleteModule 78
SOMError 79
somFree 81
SOMInitModule 83
SOMLoadModule 86
SOMMalloc 88
SOMOutCharRoutine 89
SOMRealloc 91
SOM_Assert 92
SOM_ClassLibrary 94
SOM_CreateLocalEnvironment Macro 96
SOM_DestroyLocalEnvironment 98
SOM_Error 100
SOM_Expect 101
SOM_GetClass 103
SOM_InitEnvironment 105
SOM_MainProgram 107
SOM_NoTrace 109
SOM_ParentNumResolve 110
SOM_Resolve 112
SOM_ResolveNoCheck 114
SOM_SubstituteClass 116
SOM_Test 117
SOM_TestC 119
SOM_UninitEnvironment 121
SOM_WarnMsg 123
SOMClass 124
somAddDynamicMethod 128
somAllocate 131
somCheckVersion 133

somClassReady 135
somDeallocate 136
somDescendedFrom 138
somFindMethod 140
somFindMethodOk 143
somFindSMethod 146
somFindSMethodOk 148
somGetInstancePartSize 150
somGetInstanceSize 152
somGetInstanceToken 154
somGetMemberToken 156
somGetMethodData 158
somGetMethodDescriptor 160
somGetMethodIndex 162
somGetMethodToken 164
somGetname 166
somGetNthMethodData 168
somGetNthMethodInfo 170
somGetNumMethods 172
somGetNumStaticMethods 174
somGetParents 176
somGetVersionNumbers 178
somLookupMethod 180
somNew 182
somNewNoInit 184
somRenew 186
somRenewNoInit 189
somRenewNoInitNoZero 192
somRenewNoZero 195
somSupportsMethod 198
SOMClassMgr 200
somClassFromId 202
somFindClass 204
somFindClsInFile 207
somGetInitFunction 210
somGetRelatedClasses 212
somLoadClassFile 214
somLocateClassFile 216
somMergeInto 218
somRegisterClass 221
somSubstituteClass 222
somUnloadClassFile 224
somUnregisterClass 226

SOMObject . . . 228
somCastObj . . . 230
somDefaultAssign . . . 232
somDefaultConstAssign . . . 234
somDefaultConstCopyInit . . . 236
somDefaultCopyInit . . . 238
somDefaultInit . . . 240
somDestruct . . . 243
somDispatch . . . 246
somClassDispatch . . . 250
somDumpSelf . . . 255
somDumpSelfInt . . . 256
somFree . . . 259
somGetClass . . . 261
somGetClassName . . . 263
somGetSize . . . 265
somInit . . . 267
somIsA . . . 270
somIsInstanceOf . . . 272
somPrintSelf . . . 274
somResetObj . . . 276
somRespondsTo . . . 278
somUninit . . . 280

Chapter 2. DSOM Framework Reference . . . 283
get_next_response . . . 285
ORBfree . . . 287
send_multiple_requests . . . 289
somdExceptionFree . . . 291
SOMD_Init . . . 293
SOMD_NoORBfree . . . 295
SOMD_RegisterCallback . . . 297
SOMD_Uninit . . . 299
Context_delete Macro . . . 301
Request_delete Macro . . . 303
BOA . . . 305
change_implementation . . . 307
create . . . 309
deactivate_impl . . . 312
deactivate_obj . . . 314
dispose . . . 316
get_id . . . 318
get_principal . . . 320

impl_is_ready . . . 322
obj_is_ready . . . 324
set_exception . . . 326
Context . . . 328
create_child . . . 329
delete_values . . . 331
destroy . . . 333
get_values . . . 335
set_one_value . . . 337
set_values . . . 339
ImplementationDef . . . 341
ImplRepository . . . 343
add_class_to_impldef . . . 344
add_impldef . . . 346
delete_impldef . . . 348
find_all_impldefs . . . 350
find_classes_by_impldef . . . 352
find_impldef . . . 354
find_impldef_by_alias . . . 356
find_impldef_by_class . . . 358
remove_class_from_all . . . 360
remove_class_from_impldef . . . 362
update_impldef . . . 364
NVList . . . 366
add_item . . . 367
free . . . 369
free_memory . . . 371
get_count . . . 373
get_item . . . 375
set_item . . . 378
ObjectMgr . . . 381
somdDestroyObject . . . 383
somdGetIdFromObject . . . 385
somdGetObjectFromId . . . 387
somdNewObject . . . 389
somdReleaseObject . . . 391
ORB . . . 393
create_list . . . 394
create_operation_list . . . 396
get_default_context . . . 398
object_to_string . . . 400
string_to_object . . . 402
Request . . . 405

add_arg . . . 406
destroy . . . 409
get_response . . . 411
invoke . . . 413
send . . . 417
SOMDClientProxy . . . 419
somdProxyFree . . . 421
somdProxyGetClass . . . 423
somdProxyGetClassName . . . 425
somdReleaseResources . . . 427
somdTargetFree . . . 429
somdTargetGetClass . . . 431
somdTargetGetClassName . . . 433
SOMDObject . . . 435
create_request . . . 437
create_request_args . . . 441
duplicate . . . 445
get_implementation . . . 447
get_interface . . . 449
is_constant . . . 451
is_nil . . . 453
is_proxy . . . 455
is_SOM_ref . . . 457
release . . . 459
SOMDObjectMgr . . . 461
somdFindAnyServerByClass . . . 463
somdFindServer . . . 465
somdFindServerByName . . . 467
somdFindServersByClass . . . 469
SOMDServer . . . 471
somdCreateObj . . . 472
somdDeleteObj . . . 474
somdDispatchMethod . . . 476
somdGetClassObj . . . 478
somdObjReferencesCached . . . 480
somdRefFromSOMObj . . . 482
somdSOMObjFromRef . . . 484
SOMDServerMgr . . . 486
somdDisableServer . . . 487
somdEnableServer . . . 489
somdIsServerEnabled . . . 491
somdListServer . . . 493
somdRestartServer . . . 495

somdShutdownServer . . . 497
somdStartServer . . . 499
SOMOA . . . 501
activate_impl_failed . . . 503
change_id . . . 505
create_constant . . . 507
create_SOM_ref . . . 509
execute_next_request . . . 511
execute_request_loop . . . 513
get_SOM_object . . . 515

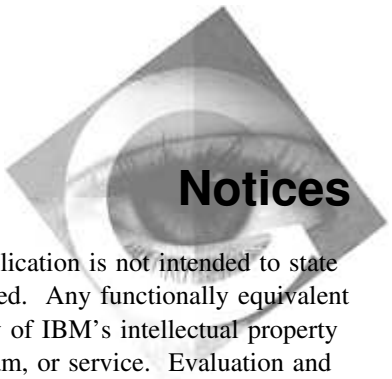
Chapter 3. Interface Repository Framework Reference . . . 517
AttributeDef . . . 519
ConstantDef . . . 521
Contained . . . 523
describe . . . 525
within . . . 528
Container . . . 530
contents . . . 531
describe_contents . . . 534
lookup_name . . . 537
ExceptionDef . . . 540
InterfaceDef . . . 542
describe_interface . . . 544
ModuleDef . . . 546
OperationDef . . . 547
ParameterDef . . . 549
Repository . . . 551
lookup_id . . . 552
lookup_modifier . . . 554
release_cache . . . 556
TypeDef . . . 558
TypeCode_alignment . . . 560
TypeCode_copy . . . 561
TypeCode_equal . . . 563
TypeCode_free . . . 565
TypeCode_kind . . . 567
TypeCodeNew . . . 572
TypeCode_param_count . . . 576
TypeCode_parameter . . . 577
TypeCode_print . . . 580
TypeCode_setAlignment . . . 582
TypeCode_size . . . 583

Chapter 4. Metaclass Framework Reference 585
SOMMBeforeAfter Metaclass 587
sommAfterMethod 588
sommBeforeMethod 591
SOMMSingleInstance 594
sommGetSingleInstance 595
SOMMTraced Metaclass 597

Chapter 5. Event Management Framework Reference 599
SOMEClientEvent 601
somevGetEventClientData 602
somevGetEventClientType 603
somevSetEventClientData 604
somevSetEventClientType 605
SOMEEMan 606
someChangeRegData 608
someGetEManSem 610
someProcessEvent 612
someProcessEvents 614
someQueueEvent 616
someRegister 618
someRegisterEv 620
someRegisterProc 622
someReleaseEManSem 624
someShutdown 626
someUnRegister 628
SOMEEMRegisterData 630
someClearRegData 631
someSetRegDataClientType 632
someSetRegDataEventMask 633
someSetRegDataSink 635
someSetRegDataSinkMask 636
someSetRegDataTimerCount 638
someSetRegDataTimerInterval 640
SOMEEvent 642
somevGetEventTime 643
somevGetEventType 644
somevSetEventTime 645
somevSetEventType 646
SOMESinkEvent 647
somevGetEventSink 648
somevSetEventSink 649
SOMETimerEvent 650

somevGetEventInterval 651
somevSetEventInterval 652
SOMEWorkProcEvent 653

Index 655



Notices

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

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

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

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

Programming Interface Information

This book is intended to help you create programs using VisualAge for C++. It primarily documents the General-Use Programming Interface and Associated Guidance Information provided by the VisualAge for C++ product.

General-Use programming interfaces allow the customer to write programs that obtain the services of the VisualAge for C++ compiler, debugger, browser, execution trace analyzer, visual builder, editor, data access frameworks, and class libraries.

However, this book also documents Diagnosis, Modification, and Tuning Information. Diagnosis, Modification, and Tuning Information is provided to help you debug your programs.

© Copyright IBM Corp. 1994, 1996

Warning: Do not use this Diagnosis, Modification, and Tuning Information as a programming interface because it is subject to change.

Diagnosis, Modification, and Tuning Information is identified where it occurs by an introductory statement to a chapter or section.

Trademarks and Service Marks

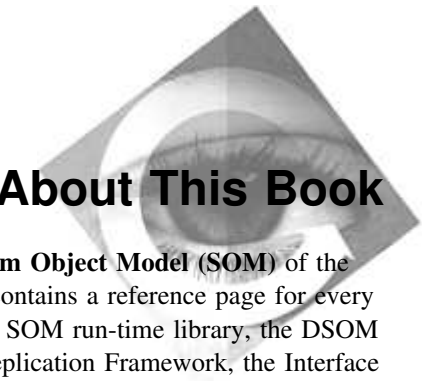
The following terms are trademarks of the International Business Machines Corporation in the United States or other countries or both:

AIX
IBM
IBMLink
OS/2
SOMobjects
System Object Model
VisualAge

Windows is a trademark of Microsoft Corporation.

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

IBM's VisualAge products and services are not associated with or sponsored by Visual Edge Software, Ltd..



About This Book

This book provides reference material for the **System Object Model (SOM)** of the **SOMobjects Developer Toolkit**. In particular, it contains a reference page for every class, method, function, and macro provided by the SOM run-time library, the DSOM run-time library, the Persistence Framework, the Replication Framework, the Interface Repository Framework, and the Event Management Framework. It also includes documentation of the utility metaclasses provided by the SOMobjects Developer Toolkit, and each of their methods.

Note: The Windows NT and Windows 95 version of SOM/DSOM that is part of this VisualAge product corresponds to SOM/DSOM version 2.1.

The term "SOMobjects Developer Toolkit" is another name for the functionality included with VisualAge for C++ Version 3.5. The term "SOMobject Base Toolkit" refers to a subset of this functionality and has been released by itself on other platforms. The version of SOM included with this release contains all of the functionality of the "SOMobjects Developer Toolkit."

In addition to this book, refer to the *SOM Programming Guide* for introductory information.

How This Book Is Organized

At the highest level, this book is organized by framework. Within each framework, the reference pages describe the classes in alphabetical order, with the methods of each class given in alphabetical order following their corresponding class. Similarly, related functions and SOM macros are given in separate alphabetical sequences in the corresponding section. The reference page for a SOM **class** contains the following topics:

Description:	A description of the class.
File Stem:	The file stem for the class's IDL interface specification (.idl) file and its usage binding (.h/.xh) files.
Base Class:	The class's direct base (parent) classes.
Ancestor Classes:	The class's ancestor (indirect base) classes.
Metaclass:	The class's metaclass.
New Methods:	The names of the methods that the class introduces (grouped roughly according to purpose). Each new method is documented on a separate reference page.
Overriding Methods:	The names of the methods that the class overrides from ancestor classes.

© Copyright IBM Corp. 1994, 1996

The reference page for a **method** of a SOM class contains the following topics:

Purpose:	The purpose of the method in brief.
Syntax:	The method's C/C++ procedure prototype (which includes the method procedure's return type and the names and types of its parameters). The in/out/inout keywords associated with each of the method's parameters in the method's IDL declaration are also shown. These keywords are shown for information only; they are not actually present in the method procedure prototype.
Description:	A description of the method's use.
Parameters:	A description of each of the method procedure's parameters.
Return Value:	A description of the method's return value.
Example:	An example of using or overriding the method, if available. Although methods of SOM classes are language neutral (i.e., they can be invoked from any programming language that can use SOM), the examples given here are written in C.
Original Class:	The name of the class that introduces the method (the class is documented separately in this book).
Related Information:	Related methods and functions (and macros, for the SOM kernel) that can be found in this book.

The reference page for a **function** has the following topics:

Purpose:	The purpose of the function in brief.
Syntax:	The function's prototype (which includes the return type and the names and types of the parameters).
Description:	A description of the function's use.
Parameters:	A description of each of the function's parameters.
Return Value:	A description of the function's return value.
Example:	An example of using the function, if available.
Related Information:	Related methods and functions (and macros, for the SOM kernel) that can be found in this book.

The reference page for a **macro** has the following fields:

Purpose:	The purpose of the macro in brief.
Syntax:	The syntax for invoking the macro .
Description:	A description of the macro's use .
Parameters:	A description of each of the macro's parameters.
Expansion:	A description of the macro's expansion (although the exact code expansion is not always given) .
Example:	An example of invoking the macro, if available.
Related Information:	Related macros and functions that can be found in this book.

Who Should Use This Book

This book is for the professional programmer using the SOMobjects Developer Toolkit to build object-oriented class libraries or application programs that use SOM class libraries or the frameworks in the SOMobjects Developer Toolkit.

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.

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.

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.

How to Get Help

There are three kinds of online information available to you while you are using VisualAge for C++:

Online documents

These are complete documents, like the one you are reading now, presented online. These documents contain detailed information on the different aspects of VisualAge for C++. For your convenience, the online documents are presented in:

- Standard format (`.INF` files). See "Getting Help Inside VisualAge for C++" on page xviii for instructions on opening standard format documents from inside VisualAge for C++. See "Getting Help from the Command Line" on page xviii for instructions on opening standard format documents from the command line. For a list of the VisualAge for C++ documents that are available in standard format, see "Online Documents Available in VisualAge for C++" on page xix.

Contextual help

Contextual help is available throughout VisualAge for C++. This help tells you all about the elements that you see in the interface, including menus, entry fields, and pushbuttons.

How Do I **help**

Many of the common tasks that you want to perform with VisualAge for C++ are described in *How Do I* help. The *How Do I* help for a task gives you step-by-step instructions for completing the task. There is overall *How Do I* help for VisualAge for C++, as well as individual task lists for each of its components.

Getting Help Inside VisualAge for C++

All three kinds of help are available directly within the VisualAge for C++ interface:

- To get general contextual help for the component of VisualAge for C++ that you are using, press **F1** anywhere in the window.
- To get contextual help on a particular menu, menu item, or button, highlight the element and press **F1**.
- To get access to all of the help information that is available to you in a particular window, click on **Help** in the menu bar at the top of the window. This menu includes the following selections:
  - **Help Index**, an alphabetical list of all of the help topics that are available from this window
  - **General Help**, overall help for the window
  - **Using Help**, general information about the help facility
  - **How Do I...**, the How Do I help for the component
  - **Product Information**, a dialog that shows the level of VisualAge for C++ being used

  In addition, there are selections that let you open all of online documents that are available in VisualAge for C++.
- To get detailed information, open the **Online Information** notebook in the VisualAge for C++ folder. In this notebook you will find tabs for **Guides**, **References**, and **How Do I** help. Each page in the notebook lists a variety of online documents that describe, in detail, the different aspects of VisualAge for C++. To open a particular online document, select the radio button for the document, and click on the **View** pushbutton.

Getting Help from the Command Line

If you want, you can look at the online documents by issuing the `iview` command. The installation routine stores the online document files in the `\IBMCPPW\HELP` directory. To view the *Language Reference*, for example, make `C:\IBMCPPW\HELP` your current directory (substituting the drive where you installed VisualAge for C++ for `C:`) and enter the following command:

```
IVIEW CPPLNG.INF
```

If you want to get information on a specific topic, you can specify a word or a series of words after the file name. If the words appear in an entry in the table of contents or the index, the online document is opened to the associated section. For example, if you want to read the section on operator precedence in the *Language Reference*, you can enter the following command:

```
IVIEW CPPLNG.INF OPERATOR PRECEDENCE
```

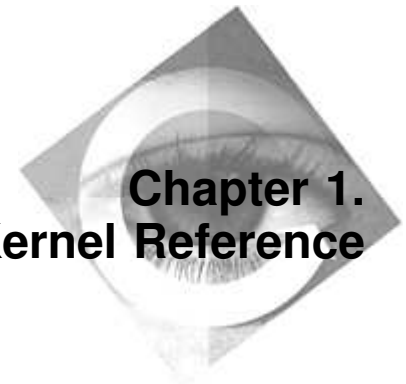
Getting Help for a Keyword or Construct

If you are editing a file using the Editor, you can get help for a keyword or construct by moving the cursor to the word and pressing **`Ctrl+H`**. In the other tools, you can get help for a keyword or construct by highlighting the word and pressing **`Ctrl+H`**.

Online Documents Available in VisualAge for C++

The following documents are available in standard format:

Building VisualAge for C++ Parts for Fun and Profit	*Open Class Library Reference*
C Library Reference	*Open Class Library User's Guide*
Editor Command Reference	*Programming Guide*
Frequently Asked Questions	*SOM Programming Guide*
Installation Guide and Product Overview	*SOM Programming Reference*
IPF User's Guide	*User's Guide*
IPF Programmer's Guide and Reference	*Visual Builder User's Guide*
Language Reference	*Visual Builder Parts Reference*



Chapter 1. SOM Kernel Reference

© Copyright IBM Corp. 1994, 1996

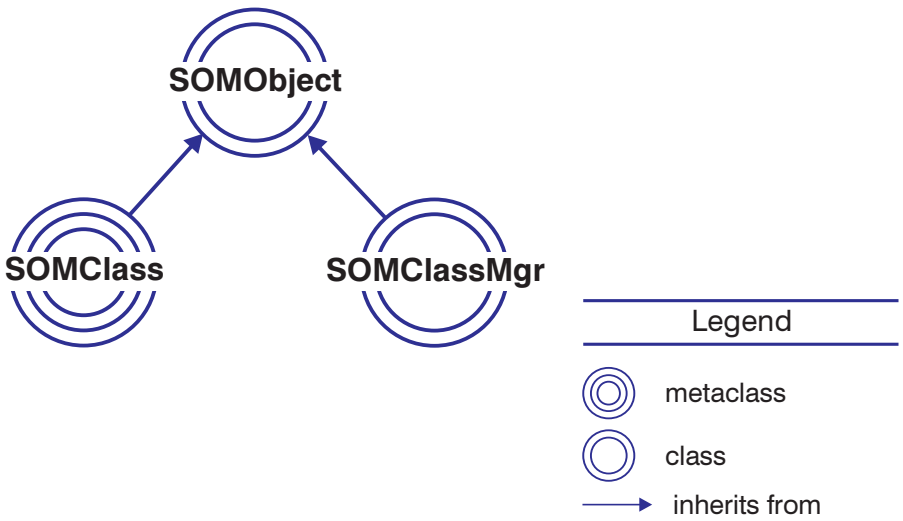


Figure 1. SOM Kernel Class Organization

somApply

This function invokes an apply stub. Apply stubs are never invoked directly by SOM users. The **somApply** function must be used instead.

Syntax

```
boolean somApply  (SOMObject  objPtr, somToken  *retVal,
                   somMethodDataPtr  mdPtr, va_list  args)
```

Parameters

objPtr (SOMObject)
A pointer to the object on which the method procedure is to be invoked.

retVal (somToken *)
A pointer to the memory region into which the result returned by the method procedure is to be copied. This pointer **cannot** be null (even in the case of method procedures whose returned result is void).

mdPtr (somMethodDataPtr)
A pointer to the **somMethodData** structure that describes the method whose procedure is to be executed by the apply stub.

args (va_list)
A **va_list** that contains the arguments for the method procedure. The first entry of the **va_list** must be *objPtr*. Furthermore, all arguments on the **va_list** must appear in widened form, as defined by ANSI C. For example, a **float** must appear as a **double**, and a **char** and a **short** must appear as the **int** data type. The SOM API for **va_list** construction ensures this.

Returns

rc (boolean)

Remarks

This function provides a single uniform interface through which it is possible to call any method procedure. The interface is based on the caller passing: the object to which the method procedure is to be applied; a return address for the method result; a **somMethodDataPtr** indicating the desired method procedure; and an ANSI standard

va_list structure containing the method procedure arguments. Different method procedures expect different argument types and return different result types, so the purpose of this function is to select an *apply stub* appropriate for the specific method involved, according to the supplied method data, and then call this apply stub. The apply stub removes the arguments from the **va_list**, calls the method procedure with these arguments, accepts the returned result, and then copies this result to the location pointed to by *retVal*.

The method procedure used by the apply stub is determined by the content of the **somMethodData** structure pointed to by *mdPtr*. The class methods **somGetMethodData** and **somGetNthMethodData** are used to load a **somMethodData** structure. These methods resolve static method procedures based on the receiving class's instance method table.

The SOM API requires that information necessary for selecting an apply stub be provided when a new method is registered with its introducing class (via the methods **somAddStaticMethod** or **somAddDynamicMethod**). This is required because SOM itself needs apply stubs when dispatch method resolution is used. C and C++ implementation bindings for SOM classes support this requirement, but SOM does not terminate execution if this requirement is not met by a class implementor. Thus, it is possible that there may be methods for which this function cannot select an appropriate apply stub. If an apply stub can be selected, then **somApply** performs as described above, and a TRUE value is returned; otherwise FALSE is returned.

Related Information

Data Structures

- **SOMObject** (somobj.idl)
- **somMethodData** (somapi.h)
- **somToken** (sombtype.h)
- **somMethodPtr** (sombtype.h)
- **va_list** (stdarg.h)

Methods

- **somGetMethodData**
- **somGetNthMethodData**
- **somAddDynamicMethod** (somcls.idl)

Example Code

```
#include <somcls.xh>
#include <string.h>
#include <stdarg.h>
main()
{
    somVaBuf vb;
    va_list args;
    string result;
    SOMClass *scObj;
    somMethodData md;

    somEnvironmentNew();    /* Init environment */
    scObj = _SOMClass;   /* The SOMClass object */

    scObj->somGetMethodData(somIdFromString("somGetName"), &md);
    vb = (somVaBuf)somVaBuf_create(NULL, 0);
    somVaBuf_add(vb, (char *)&scObj, tk_ulong);
    somVaBuf_get_valist(vb, &args);

    somApply(scObj, (somToken*)&result, &md, args);
    SOM_Assert(!strcmp(result, "SOMClass"), SOM_Fatal);
    /* result is "SOMClass" */
}
```

somBuildClass

This function automates the process of building a new SOM class object.

Syntax

```
SOMClass somBuildClass  (unsigned long inheritVars,
                         somStaticClassInfoPtr sciPtr,
                         long majorVersion, long minorVersion)
```

Parameters

inheritVars (unsigned long)
A bit mask that determines inheritance from parent classes. A mask containing all ones is an appropriate default.

sciPtr (somStaticClassInfoPtr)
A pointer to a structure holding static class information.

majorVersion (long)
The major version number for the class.

minorVersion (long)
The minor version number for the class.

Returns

rc (SOMClass)
A pointer to a class object.

Remarks

This function accepts declarative information defining a new class that is be built, and performs the activities required to build and register a correctly functioning class object. The C and C++ implementation bindings use this function to create class objects.

Related Information

Data Structures

- **somStaticClassInfo (somapi.h)**

Example Code

```
See any .ih or .xih implementation binding file for
details on construction of the required data
structures.
```

somCheckId

This function registers a som ID.

Syntax

```
somId somCheckId  (somId  id)
```

Parameters

id (somId)
The **somId** to be registered.

Returns

rc (somId)

The registered **somId**.

Remarks

This function registers a SOM ID and converts it into an internal representation. The input SOM ID is returned. If the ID is already registered, this function has no effect.

Related Information

Data Structures

- **somId (sombtype.h)**

Methods

- **somRegisterId**
- **somFromString**
- **somStringFromId**
- **comCompareIds**
- **somTotalRegIds**
- **somSetExpectedIds**
- **somUniqueKey**

somClassResolve

This function obtains a pointer to the procedure that implements a static method for instances of a particular SOM class.

Syntax

```
somMethodPtr somClassResolve   (SOMClass  cls, somMToken  mToken)
```

Parameters

cls (SOMClass)
A pointer to the class object whose instance method procedure is required.

mToken (somMToken)
The method token for the method to be resolved. The SOM API requires that if the class "XYZ" introduces the static method "foo", then the method token for "foo" is found in the class data structure for "XYZ" (called XYZClassData) in the structure member named "foo" (i.e., at XYZClassData.foo). Method tokens can also be obtained using the **somGetMethodToken** method. A pointer to the class object whose instance method procedure is required.

Returns

rc (somMethodPtr)

A pointer to the **somMethodProc** (procedure) that implements the specified method for the specified class of SOM object.

Remarks

This function is used to obtain a pointer to the procedure that implements the specified method for instances of the specified SOM class. The returned procedure pointer can then be used to invoke the method. The somClassResolve function is used to support "casted" method calls, in which a method is resolved with respect to a specified class rather than the class of which an object is a direct instance. The **somClassResolve** function can only be used to obtain a method procedure for a static method (a method declared in an IDL specification for a class); dynamic methods do not have method tokens.

The SOM language usage bindings for C and C++ do not support casted method calls, so this function must be used directly to achieve this functionality. Whenever

using SOM method procedure pointers, it is necessary to indicate the use of system linkage to the compiler. The way this is done depends on the compiler and the system being used. However, C and C++ usage bindings provide an appropriate typedef for this purpose. The name of the typedef is based on the name of the class that introduces the method, as illustrated in the example below.

Related Information

Data Structures

- **somMethodPtr (sombtype.h)**
- **SOMClass (somcls.idl)**
- **somMToken (somapi.h)**

Functions

- **somResolveByName**
- **somParentResolve**
- **somParentNumResolve**
- **somResolve**

Methods

- **somDispatch**
- **somClassDispatch**
- **somFindMethod**
- **somFindMethodOk**
- **somGetMethodToken**

Macros

- **SOM_Resolve**
- **SOM_ResolveNoCheck**

Example Code

```
// SOM IDL for class A and class B
#include <somobj.idl>
module scrExample {
  interface A : SOMObject { void foo(); implementation
{ callstyle=oidl;};};
  interface B : A  {  implementation { foo: override;};};
};

// Example C++ program to implement and test module scrExample
#define SOM_Module_screxample_Source
#include <scrExample.xih>
#include <stdio.h>

SOM_Scope void SOMLINK scrExample_Afoo(scrExample_A *somSelf);
{  printf("1\n");}

SOM_Scope void SOMLINK scrExample_Bfoo(scrExample_B *somSelf);
{ printf("2\n");}

main()
{
  scrExample_B  *objPtr = new scrExample_B;

// This prints 2
objPtr->foo();

// This prints 1
   ((somTD_scrExample_A_foo) /* A necessary method procedure cast */
     somClassResolve(
              ,_scrExample_A, // the A class object
              scrExample_AClassData.foo) // the foo method token
     ) /* end of method procedure expression */
     (objPtr); /* method arguments */

// This prints 2
   ((somTD_scrExample_A_foo) /* A necessary method procedure cast */
     somClassResolve(
              ,_scrExample_B, // the B class object
              scrExample_AClassData.foo) // the foo method token
     ) /* end of method procedure expression */
     (objPtr); /* method arguments */

}
```

somCompareIds

This function determines whether two SOM IDs represent the same string.

Syntax

```
int somCompareIds  (somId  id1, somId  id2)
```

Parameters

id1 (somId)
The first SOM ID to be compared.

id2 (somId)
The second SOM ID to be compared.

Returns

rc (int)

1 Returns 1 if the two input IDs represent strings that are equal.
0 Returns 0 if the two input IDs represent strings that are not equal.

Remarks

This function returns 1 if the two input IDs represent strings that are equal; otherwise, it returns 0.

Related Information

Data Structures

- **somId (sombtype.h)**

Functions

- **somCheckId**
- **somRegisterId**
- **somIdFromString**
- **somStringFromId**
- **somTotalRegIds**
- **somSetExpectedIds**
- **somUniqueKey**

Example Code

```
#include <som.h>
main()
{
   somId id1, id2, id3;

   somEnvironmentNew();
   id1 = somIdFromString("this");
   id2 = somIdFromString("that");
   id3 = somIdFromString("this");

   SOM_Test(somCompareIds(id1, id3));
   SOM_Test(!somCompareIds(id1, id2));
}
```

somDataResolve

This function accesses instance data within an object.

Syntax

```
somToken somDataResolve  (SOMObject  obj, somDToken  dToken)
```

Parameters

obj (SOMObject)
A pointer to the object whose instance data is required.

dToken (somDToken)
A data token for the required instance data. The SOM API specifies that the data token for accessing the instance data introduced by a class is found in the instanceDataToken component of the auxiliary class data structure for that class. The example below illustrates this.

Returns

rc (somToken)

A **somToken** (that is, a pointer) that points to the data in obj identified by the dToken. If *obj* does not contain the requested data identified by *dToken*, **somDataResolve** generates a runtime error.

Remarks

The **somDataResolve** function is used to access instance data within an object. This function is of use primarily to class implementors (rather than class clients) who are not using the SOM C or C++ language bindings.

For C or C++ programmers with access to the C or C++ implementation bindings for a class, instance data can be accessed using the GetData macro (which expands to a usage of somDataResolve).

Related Information

Data Structures

- **somToken (sombtype.h)**
- **SOMObject (somobj.idl)**

- **somDToken (somapi.h)**

Example Code

The following C/C++ expression evaluates to the address of the instance data introduced by class "XYZ" within the object "obj". This assumes that obj points to an instance of "XYZ" or a subclass of "XYZ".

```
#include <som.h>
somDataResolve(obj, XYZCClassData.instanceDataToken);
```

somDataResolveChk

This function accesses instance data within an object.

Syntax

```
somToken somDataResolveChk   (SOMObject  obj, somDToken  dToken)
```

Parameters

obj (SOMObject)
A pointer to the object whose instance data is required.

dToken (somDToken)
A data token for the required instance data. The SOM API specifies that the data token for accessing the instance data introduced by a class is found in the instanceDataToken component of the auxiliary class data structure for that class. The example below illustrates this.

Returns

rc (somToken)

A **somToken** (that is, a pointer) that points to the data in obj identified by the dToken. If *obj* does not contain the requested data identified by *dToken*, **somDataResolveChk** returns NULL.

Remarks

The **somDataResolveChk** function is used to access instance data within an object. This function is of use primarily to class implementors (rather than class clients) who are not using the SOM C or C++ language bindings.

For C or C++ programmers with access to the C or C++ implementation bindings for a class, instance data can be accessed using the GetData macro (which expands to a usage of somDataResolve).

Related Information

Data Structures

- **somToken (sombtype.h)**
- **SOMObject (somobj.idl)**

- **somDToken (somapi.h)**

Example Code

The following C/C++ expression evaluates to the address of the instance data introduced by class “XYZ” within the object “obj”. This assumes that obj points to an instance of “XYZ” or a subclass of “XYZ”.

```
#include <som.h>
somDataResolveChk(obj, XYZCClassData.instanceDataToken);
```

somEnvironmentEnd

This function provides general cleanup for applications.

Syntax

```
void somEnvironmentEnd ()
```

Parameters

None.

Returns

rc (void)

Remarks

This function is a general cleanup function that must be called by all Windows applications before exiting. AIX and OS/2 programs may also invoke this function, but it is not required on these systems because all necessary SOM cleanup is performed by the operating system during program termination.

A convenience macro, **SOM_MainProgram**, which usually appears at the beginning of each application, adds this function to the "atexit" list. If the "atexit" mechanism does not work reliably with your compiler, or if you know that your program bypasses the normal program termination sequence, you should insert an explicit call to this function at the point where your main program exits. (All main programs for Windows must begin either with the **SOM_MainProgram** macro or with a call to the **somMainProgram** function.)

Related Information

Macros

- **SOM_MainProgram**

somEnvironmentNew

This function initializes the SOM runtime environment.

Syntax

```
SOMClassMgr somEnvironmentNew ()
```

Parameters

None.

Returns

rc (SOMClassMgr)

A pointer to the single class manager object active at run time. This class manager can be referred by the global variable *SOMClassMgrObject*.

Remarks

This function creates the four primitive SOM objects (*SOMObject, SOMClass, SOMClassMg*r, and *SOMClassMgrObject*) and initializes global variables used by the SOM run-time environment. This function must be called before using any other SOM functions or methods (with the exception of **somSetExpectedIds**). If the SOM run-time environment has already been initialized, calling this function has no harmful effect.

Although this function must be called before using other SOM functions or methods, it needn't always be called explicitly, because the **New** macros, the **Renew** macros, the **new** operator, and the **NewClass** procedures defined by the SOM C and C++ language bindings call **somEnvironmentNew** if needed.

Related Information

Functions

- **somExceptionId**
- **somExceptionValue**
- **somSetException**
- **somGetGlobalEnvironment**

somEnvironmentNew

Example Code

```
somEnvironmentNew();
```

somExceptionFree

This function frees the memory held by the exception structure within an **Environment** structure.

Syntax

```
void somExceptionFree  (Environment  *env)
```

Parameters

env (Environment *)
A pointer to the **Environment** whose exception information is to be freed.

Returns

rc (void)

Remarks

This function frees the memory held by the exception structure within an **Environment** structure.

Related Information

Data Structures

- **Environment (somcorba.h)**

Functions

- **somExceptionId**
- **somExceptionValue**
- **somSetException**
- **somGetGlobalEnvironment**
- **somdExceptionFree (DSOM function)**

somExceptionFree

Example Code

```
See function somSetException.
```

somExceptionId

This function gets the name of the exception contained in an **Environment** structure.

Syntax

```
string somExceptionId   (Environment  *env)
```

Parameters

env (Environment *)
A pointer to an **Environment** structure containing an exception.

Returns

rc (string)

string

Returns the name of the exception contained in the specified **Environment** structure, as a string.

Remarks

This function returns the name of the exception contained in the specified **Environment** structure.

Related Information

Data Structures

- **string (somcorba.h)**
- **Environment (somcorba.h)**

Functions

- **somExceptionValue**
- **somExceptionFree**
- **somSetException**
- **somGetGlobalEnvironment**

somExceptionId

Example Code

```
See function somSetException.
```

somExceptionValue

This function gets the value of the exception contained in an **Environment** structure.

Syntax

```
somToken somExceptionValue  (Environment  *env)
```

Parameters

env (Environment *)
A pointer to an **Environment** structure containing an exception.

Returns

rc (somToken)
Returns a pointer to the value of the exception contained in the specified **Environment** structure.

Remarks

This function returns the value of the exception contained in the specified **Environment** structure.

Related Information

Data Structures

- **somToken (sombtype.h)**
- **Environment (somcorba.h)**

Functions

- **somExceptionId**
- **somExceptionFree**
- **somSetException**
- **somGetGlobalEnvironment**
- **somSetGlobalEnvironment**

somExceptionValue

Example Code

```
See function somSetException.
```

somGetGlobalEnvironment

This function returns a pointer to the current global **Environment** structure.

Syntax

```
Environment * somGetGlobalEnvironment ()
```

Parameters

None.

Returns

rc (Environment *)

A pointer to the current global **Environment** structure.

Remarks

This function returns a pointer to the current global **Environment** structure. This structure can be passed to methods that require an **(Environment *)** argument. The caller can determine if the called method has raised an exception by testing whether

```
ev->exception._major != NO_EXCEPTION
```

If an exception has been raised, the caller can retrieve the name and value of the exception using the **somExceptionId** and **somExceptionValue** methods.

Related Information

Data Structures

- **Environment (somcorba.h)**

Functions

- **somExceptionId**
- **somExceptionValue**
- **somExceptionFree**
- **somSetException**

somGetGlobalEnvironment

Example Code

```
See function somSetException.
```

somIdFromString

This function returns the SOM ID corresponding to a given text string.

Syntax

```
somId somIdFromString  (string aString)
```

Parameters

aString (string)
The string to be converted to a SOM ID.

Returns

rc (somId)

Returns the SOM ID corresponding to the given text string.

Remarks

This function returns the SOM ID that corresponds to a given text string.

Ownership of the **somId** returned by this function passes to the caller, which has the responsibility to subsequently free the **somId** using **SOMFree**.

Related Information

Data Structures

- **somId (sombtype.h)**
- **string (somcorba.h)**

Functions

- **somCheckId**
- **somRegisterId**
- **somStringFromId**
- **somCompareIds**
- **somTotalRegIds**
- **somSetExpectedIds**
- **somUniqueKey**

somIsObj

This function is a failsafe routine to determine whether a pointer references a valid SOM object.

Syntax

```
boolean somIsObj   (somToken  memPtr)
```

Parameters

memPtr (somToken)
A **somToken** (a pointer) to be checked.

Returns

rc (boolean)

1 Returns 1 if *obj* is a pointer to a valid SOM object.
0 Returns 0 if *obj* is not a pointer to an valid SOM object.

Remarks

This function returns 1 if its argument is a pointer to a valid SOM object, or returns 0 otherwise. The function handles address faults, and does extensive consistency checking to guarantee a correct result.

Related Information

Data Structures

- **boolean** (somcorba.h)
- **somToken** (sombtype.h)

Example Code

```
#include <stdio.h>
#include <som.xh>

void example(void *memPtr)
{
   if (!somIsObj(memPtr))
     printf("memPtr is not a valid SOM object.\n");
   else
     printf("memPtr points to an object of class %s\n",
             ((SOMObject *)memPtr)->somGetClassName());
}
```

somLPrintf

This function prints a formatted string in the manner of the C printf function, at the specified indentation level.

Syntax

```
long somLPrintf  (long  level, string  fmt, . . .)
```

Parameters

level (long)
The level at which output is to be placed.

fmt (string)
The format string to be output.

. . .
The values to be substituted into the format string.

Returns

rc (long)
Returns the number of characters written.

Remarks

This function prints a formatted string using SOMOutCharRoutine, in the same manner as the C printf function. The implementation of SOMOutCharRoutine determines the destination of the output, while the C printf function is always directed to stdout. (The default output destination for SOMOutCharRoutine is stdout also, but this can be modified by the user.) The output is prefixed at the indicated level, by preceding it with 2*level spaces.

Related Information

Data Structures

- **string (somcorba.h)**

Functions

- **somVprintf**
- **somPrefixLevel**

- **somPrintf**
- **SOMOutCharRoutine**

Example Code

```
#include <somobj.h>
somLPrintf(5, "The class name is %s.\n", _somGetClassName(obj));
```

somMainProgram

This function performs SOM initialization on behalf of a new program.

Syntax

```
SOMClassMgr * somMainProgram ()
```

Parameters

None.

Returns

rc (SOMClassMgr *)
A pointer to the **SOMClassMgr** object.

Remarks

This function informs SOM about the beginning of a new thread of execution (called a *task* on Windows). The SOM Kernel then performs any needed initialization, including the deferred execution of the **SOMInitModule** functions found in statically-loaded class libraries. This function must appear near the beginning of all Windows main programs, and may also be used in AIX or OS/2 programs. When used, it supersedes any need to call the **somEnvironmentNew** function.

A convenience macro, **SOM_MainProgram**, which combines the execution of this function with the scheduling of the **somEnvironmentEnd** function during normal program termination, is available for C and C++ programmers.

Related Information

Functions

- **somEnvironmentNew**
- **somEnvironmentEnd**

Macros

- **SOM_MainProgram**
- **SOM_ClassLibrary**

somParentNumResolve

This function obtains a pointer to a procedure that implements a method, given a list of method tables.

Syntax

```
somMethodPtr somParentNumResolve  (somMethodTabs  parentMtab,
                                   int  parentNum,
                                   somMToken  mToken)
```

Parameters

parentMtab (somMethodTabs)

A list of method tables for the parents of the class being implemented. The SOM API specifies that the list of parent method tables for a given class be stored in the auxiliary class data structure of the class, in the *parentMtab* component. Thus, for the class "XYZ", the parent method table list is found in location *XYZClassData.parentMtab.* Parent method table lists are available from class objects via the method call **somGetPClsMtabs**.

parentNum (int)

The position of the parent for which the method is to be resolved. The order of a class's parents is determined by the order in which they are specified in the interface statement for the class. (The first parent is number 1.)

mToken (somMToken)

The method token for the method to be resolved. The SOM API requires that if the class "XYZ" introduces the static method "foo", then the method token for "foo" is found in the class data structure for "XYZ" (called XYZClassData) in the structure member named "foo" (i.e., at XYZClassData.foo). Method tokens can also be obtained using the **somGetMethodToken** method.

Returns

rc (somMethodPtr)

A pointer to a **somMethodProc** (procedure) that implements the specified method, selected from the specified method table.

somParentNumResolve

Remarks

This function is used to make parent method calls by the C and C++ language implementation bindings. The **somParentNumResolve** function returns a pointer to a procedure for performing the specified method. This pointer is selected from the specified method table, which is intended to be the method table corresponding to a parent class.

For C and C++ programmers, the implementation bindings for SOM classes provide convenient macros for making parent method calls (the "parent_" macros).

Related Information

Data Structures

- **Data Structures**
- **somMethodPtr(sombtype.h)**
- **somMethodTabs(somapi.h)**
- **somMToken (somapi.h)**

Functions

- **somResolveByName**
- **somResolve**
- **somParentNumResolve**
- **somClassResolve**

Methods

- **somGetPClsMtab**
- **somGetPClsMtabs**
- **somGetMethodToken**

Macros

- **SOM_ParentNumResolve**
- **SOM_Resolve**
- **SOM_ResolveNoCheck**

Example Code

```
// SOM IDL for class A and class B
#include <somobj.h>
module spnrExample {
  interface A : SOMObject  { void foo();
implementation {callstyle=oidl; }; };
  interface B : A  {  implementation { foo: override; }; };
};

// Example C++ program to implement and test  module scrExample
#define SOM_Module_spnrexample_Source
#include <spnrExample.xih>
#include <stdio.h>

SOM_Scope void SOMLINK spnrExample_Afoo(spnrExample_A *somSelf);
{  printf("1\n"); }

SOM_Scope void SOMLINK spnrExample_Bfoo(spnrExample_B *somSelf);
{ printf("2\n"); }

main()
{
  spnrExample_B  *objPtr = new spnrExample_B;

// This prints  2
objPtr->foo();

// This prints 1
   ((somTD_spnrExample_A_foo)
/* This method procedure expression cast is necessary */
           somParentNumResolve(
                    ,objPtr->somGetClass()->somGetPClsMtabs(),
                    "1"
                    spnrExample_AClassData.foo) // the foo method token
           ) /* end of method procedure expression */
           (objPtr); /* method arguments */
}
```

somParentResolve

This function obtains a pointer to a procedure that implements a method, given a list of method tables. Obsolete but still supported.

Syntax

```
somMethodPtr somParentResolve  (somMethodTabs  parentMtab,
                                somToken  mToken)
```

Parameters

parentMtab (somMethodTabs)
A list of parent method tables, the first of which is the method table for the parent class for which the method is to be resolved. The SOM API specifies that the list of parent method tables for a given class be stored in the auxiliary class data structure of the class, in the *parentMtab* component. Thus, for the class "XYZ", the parent method table list is found in location XYZCClassData.parentMtab. Parent method table lists are available from class objects via the method call **somGetPClsMtabs**.

mToken (somToken)
The method token for the method to be resolved. The SOM API requires that if the class "XYZ" introduces the static method "foo", then the method token for "foo" is found in the class data structure for "XYZ" (called XYZClassData) in the structure member named "foo" (i.e., at XYZClassData.foo). Method tokens can also be obtained using the **somGetMethodToken** method.

Returns

rc (somMethodPtr)
A pointer to the **somMethodProc** (procedure) that implements the specified method, selected from the first method table.

Remarks

This function is used by old, single-parent class binaries to make parent method calls. The function is obsolete, but is still supported. This function returns a pointer to the procedure that implements the specified method. This pointer is selected from the first method table in the parentMtab list.

Related Information

Data Structures

somMethodPtr(sombtype.h) somMethodTabs (somapi.h)
somMToken(somapi.h)

Functions

- **somResolveByName**
- **somResolve**
- **somParentNumResolve**
- **somClassResolve**

Methods

- **somDispatch**
- **somClassDispatch**
- **somFindMethod**
- **somFindMethodOk**
- **somGetMethodToken**

Macros

- **SOM_Resolve**
- **SOM_ResolveNoCheck**

somPrefixLevel

This function outputs blanks to prefix a line at the indicated level.

Syntax

```
void somPrefixLevel  (long  level)
```

Parameters

level (long)
The level at which the next line of output is to start.

Returns

rc (void)

Remarks

This function outputs blanks (via the **somPrintf** function) to prefix the next line of output at the indicated level. (The number of blanks produces is 2*level.) This function is useful when overriding the **somDumpSelfInt** method, which takes the level as an argument.

Related Information

Functions

- **somPrintf**
- **somVprintf**
- **somLPrintf**
- **SOMOutCharRoutine**

Example Code

```
#include <somcls.h>
somPrefixLevel(5);
```

somPrintf

This function prints a formatted string in the manner of the C printf function.

Syntax

```
long somPrintf   (string  fmt, . . .)
```

Parameters

fmt (string)
The format string to be output.

. . .
The values to be substituted into the format string.

Returns

rc (long)

Returns the number of characters written.

Remarks

This function prints a formatted string using function **SOMOutCharRoutine**, in the same manner as the C printf function. The implementation of **SOMOutCharRoutine** determines the destination of the output, while the C printf function is always directed to stdout. (The default output destination for **SOMOutCharRoutine** is stdout also, but this can be modified by the user.)

Related Information

Functions

- **somVprintf**
- **somPrefixLevel**
- **somLPrintf**
- **SOMOutCharRoutine**

somPrintf

Example Code

```
#include <somcls.h>
somPrintf("The class name is %s.\n", _somGetClassName(obj));
```

somRegisterId

This function registers a SOM ID and determines whether or not it was previously registered.

Syntax

```
int somRegisterId  (somId  id)
```

Parameters

id (somId)

Returns

rc (int)

0 Returns 0 if the ID is already registered.
1 Returns 1 if the ID is not already registered.

Remarks

This function registers a SOM ID and converts it into an internal representation. If the ID is already registered, this function returns 0 and has no effect. Otherwise, this function returns 1.

Related Information

Data Structures

- **somId (sombtype.h).**

Functions

- **somCheckId**
- **somIdFromString**
- **somStringFromId**
- **somCompareIds**
- **somTotalRegIds**
- **somSetExpectedIds**
- **somUniqueKey**

somRegisterId

Example Code

```
#include <som.h>
static string s = "unregistered";
static somId sid = &s;
main()
{
   somEnvironmentNew();
   SOM_Test(somRegisterId(sid) == 1);
   SOM_Test(somRegisterId(somIdFromString("registered")) == 0);
}
```

somResolve

This function obtains a pointer to the procedure that implements a method for a particular SOM object.

Syntax

```
somMethodPtr somResolve  (SOMObject  obj, somMToken  mToken)
```

Parameters

obj (SOMObject)
A pointer to the object whose method procedure is required.

mToken (somMToken)
The method token for the method to be resolved. The SOM API requires that if the class "XYZ" introduces the static method "foo", then the method token for "foo" is found in the class data structure for "XYZ" (called XYZClassData) in the structure member named "foo" (i.e., at XYZClassData.foo). Method tokens can also be obtained using the **somGetMethodToken** method.

Returns

rc (somMethodPtr)
A pointer to the **somMethodProc** (procedure) that implements the specified method for the specified SOM object.

Remarks

This function returns a pointer to the procedure that implements the specified method for the specified SOM object. This pointer can then be used to invoke the method. The **somResolve** function can only be used to obtain a method procedure for a static method (one declared in an IDL or OIDL specification for a class); dynamic methods are not supported by method tokens.

For C and C++ programmers, the SOM usage bindings for SOM classes provide more convenient mechanisms for invoking methods. These bindings use the **SOM_Resolve** and **SOM_ResolveNoCheck** macros, which construct a method token expression from the class name and method name, and call **somResolve**.

somResolve

Related Information

Data Structures

- **somMethodPtr (sombtype.h)**
- **somMToken (somapi.h)**

Functions

- **somResolveByName**
- **somParentResolve,**
- **somParentNumResolve**
- **somClassResolve**

Methods

- **somDispatch**
- **somClassDispatch**
- **somFindMethod**
- **somFindMethodOk**
- **somGetMethodToken**

Macros

- **SOM_Resolve**
- **SOM_ResolveNoCheck**

Example Code

```
// SOM IDL for class A and class B
#include <somobj.idl>
module srExample {
  interface A : SOMObject  { void foo();  implementation
{ callstyle=oidl; }; };
  interface B : A  {  implementation { foo: override; }; };
};

// Example C++ program to implement and test  module scrExample
#define SOM_Module_srexample_Source
#include <srExample.ih>
#include <stdio.h>

SOM_Scope void SOMLINK srExample_Afoo(srExample_A *somSelf);
{  printf("1\n"); }

SOM_Scope void SOMLINK srExample_Bfoo(srExample_B *somSelf);
{ printf("2\n"); }

main()
{
  srExample_B  objPtr = srExample_BNew();

/* This prints  2 */
   ((somTD_srExample_A_foo)
/* this method procedure expression cast is necessary */
   somResolve(objPtr, srExample_AClassData.foo)
   )     /* end of method procedure expression */
    (objPtr);
}
```

somResolveByName

This function obtains a pointer to the procedure that implements a method for a particular SOM object.

Syntax

```
somMethodPtr somResolveByName  (SOMObject  obj,
                                string  methodName)
```

Parameters

obj (SOMObject)
A pointer to the object whose method procedure is required.

methodName (string)
A character string representing the name of the method to be resolved.

Returns

rc (somMethodPtr)

A pointer to the **somMethodProc** (procedure) that implements the specified method for the specified SOM object.

Remarks

This function is used to obtain a pointer to the procedure that implements the specified method for the specified SOM object. The returned procedure pointer can then be used to invoke the method. The C and C++ usage bindings use this function to support name-lookup methods.

This function can be used for invoking dynamic methods. However, the C and C++ usage bindings for SOM classes do not support dynamic methods, thus typedefs necessary for the use of dynamic methods are not available as with static methods. The function somApply provides an alternative mechanism for invoking dynamic methods that avoids the need for casting procedure pointers.

Assuming the static method “setSound,” is introduced by the class “Animal”, the following example will correctly invoke this method on an instance of “Animal” or one of its descendent classes.

Related Information

Data Structures

- **somMethodPtr (sombtype.h)**
- **SOMObject (somobj.idl)**
- **string (somcorba.h)**

Functions

- **somResolve**
- **somParentResolve**
- **somParentNumResolve**
- **somClassResolve**

Methods

- **somDispatch**
- **somClassDispatch**
- **somFindMethod**
- **somFindMethodOk**

Macros

- **SOM_Resolve**
- **SOM_ResolveNoCheck**

Example Code

```
#include <animal.h>
example(Animal myAnimal)
{
somTD_Animal_setSound
   setSoundProc = somResolveByName(myAnimal, "setSound");
setSoundProc(myAnimal, "Roar!");
}
```

somSetException

This function sets an exception value in an **Environment** structure.

Syntax

```
void somSetException  (Environment  *env, exception_type  major,
                       string  exceptionName, somToken  params)
```

Parameters

env (Environment *)
A pointer to the **Environment** structure in which to set the exception. This value must be either NULL or a value formerly obtained from the function **somGetGlobalEnvironment**.

major (exception_type)
major An integer representing the type of exception to set.

exceptionName (string)
The qualified name of the exception to set. The SOM Compiler defines, in the header files it generates for an interface, a constant whose value is the qualified name of each exception defined within the interface. This constant has the name "ex_<exceptionName>", where <exceptionName> is the qualified (scoped) exception name. Where unambiguous, the usage bindings also define the short form "ex_<exceptionName>", where <exceptionName> is unqualified.

params (somToken)
A pointer to an initialized exception structure value. No copy is made of this structure; hence, the caller cannot free it. The **somExceptionFree** function should be used to free the **Environment** structure that contains it.

Returns

rc (void)

Remarks

This function sets an exception value in an **Environment** structure.

Related Information

Data Structures

- **Environment**
- **exception_type**
- **string (somcorba.h)**

Methods

- **somExceptionId**
- **somExceptionValue**
- **somExceptionFree**
- **somGetGlobalEnvironment**

Example Code

somSetException

```
/* IDL declaration of class X:  */
interface X : SOMObject {
exception OUCH {long code1; long code2; };
void foo(in long arg) raises (OUCH);
};
/* implementation of foo method */
SOM_Scope void SOMLINK foo(X somSelf, Environment *ev, long arg)
{
X_OUCH *exception_params; /* X_OUCH struct is defined
                                    in X's usage bindings    */

   if (arg > 5) /* then this is a very bad error */
   {
      exception_params = (X_OUCH*)SOM_Malloc(sizeof(X_OUCH));
      exception_params->code1 = arg;
      exception_params->code2 = arg-5;
      somSetException(ev, USER_EXCEPTION, ex_X_OUCH,
                           exception_params);
      /* the Environment ev now contains an X_OUCH exception, with
      * the specified exception_params struct. The constant
      * ex_X_OUCH is defined in foo.h. Note that exception_params
      *  must be malloced.
      */
      return;
   }
...
}

main()
{
   Environment *ev;
   X x;

   somEnvironmentNew();
   x = Xnew();
   ev = somGetGlobalEnvironment();
   X_foo(x, ev, 23);
   if (ev->_major != NO_EXCEPTION) {
   printf("foo exception = %s\n", somExceptionId(ev));
   printf("code1 = %d\n",
   ((X_OUCH*) somExceptionValue(ev))->code1);
   /* finished handling exception. */
   /* free the copied id and the original X_OUCH structure: */
   somExceptionFree(ev);
   }
...
}
```

somSetExpectedIds

This function tells SOM how many unique SOM IDs a client program expects to use.

Syntax

```
void somSetExpectedIds  (unsigned  long  numIds)
```

Parameters

numIds (unsigned long)
The number of SOM IDs the client program expects to use.

Returns

rc (void)

Remarks

This function informs the SOM run-time environment how many unique SOM IDs a client program expects to use during its execution. This has the potential of slightly improving the program's space and time efficiency, if the value specified is accurate. This function, if used, must be called prior to any explicit or implicit invocation of the **somEnvironmentNew** function to have any effect.

Related Information

Functions

- **somCheckId**
- **somRegisterId**
- **somIdFromString**
- **somStringFromId**
- **somCompareIds**
- **somTotalRegIds**
- **somUniqueKey**

somSetExpectedIds

Example Code

```
#include <som.h>
somSetExpectedIds(1000);
```

somSetOutChar

This function changes the behavior of the **somPrintf** function.

Syntax

```
void somSetOutChar (somTD_SOMOutCharRoutine *outCharRtn)
```

Parameters

outCharRtn (somTD_SOMOutCharRoutine *)
A pointer to your routine that outputs a character in the way you want.

Returns

rc (void)

Remarks

This function is called to change the output character routine that **somPrintf** invokes. By default, **somPrintf** invokes a character output routine that goes to "stdout."

The execution of this function affects only the application (or thread) in which it occurs. Thus, this function is normally preferred over **SOMOutCharRoutine** for changing the output routine called by **somPrintf**, since **SOMOutCharRoutine** remains in effect for subsequent threads as well.

Some samples of this function can be found in the "somapi.h" header file.

Related Information

Functions

- **somPrintf**
- **SOMOutCharRoutine**

somStringFromId

This function returns the string that a SOM ID represents.

Syntax

```
string somStringFromId   (somId  id)
```

Parameters

id (somId)
The SOM ID for which the corresponding string is needed.

Returns

rc (string)

Returns the string that the given SOM ID represents.

Remarks

This function returns the string that a given SOM ID represents.

Related Information

Data Structures

- **string (somcorba.h)**
- **somid (sombtype.h)**

Functions

- **somCheckId**
- **somRegisterId**
- **somIdFromString,**
- **somCompareIds,**
- **somTotalRegIds**
- **somSetExpectedIds**
- **somUniqueKey**

somTotalRegIds

This function returns the total number of SOM IDs that have been registered.

Syntax

```
unsigned long somTotalRegIds ()
```

Parameters

None.

Returns

rc (unsigned long)

Returns the total number of SOM IDs that have been registered.

Remarks

This function returns the total number of SOM IDs that have been registered so far. This value can be used as a parameter to the **somSetExpectedIds** function to advise SOM about expected ID usage in later executions of a client program.

Related Information

Functions

- **somCheckId**
- **somRegisterId**
- **somIdFromString**
- **somStringFromId**
- **somCompareIds**
- **somSetExpectedIds**
- **somUniqueKey**

somTotalRegIds

Example Code

```
#include <som.h>
main()
{ int i;
  somId id;
  somEnvironmentNew();
  id = somIdFromString("abc")
  i = somTotalRegIds();
  id = somIdFromString("abc");
  SOM_Test(i == somTotalRegIds);
}
```

somUniqueKey

This function returns the unique key associated with a SOM ID.

Syntax

```
unsigned long somUniqueKey (somID id)
```

Parameters

id (somID)
The SOM ID for which the unique key is needed. An **unsigned long** representing the unique key of the specified SOM ID.

Returns

rc (unsigned long)
An **unsigned long** representing the unique key of the specified SOM ID.

Remarks

This function returns the unique key associated with a SOM ID. The unique key for a SOM ID is a number that uniquely represents the string that the SOM ID represents. The unique key for a SOM ID is the same as the unique key for another SOM ID only if the two SOM IDs represent the same string.

Related Information

Data Structures

- **somId (sombtype.h)**

Functions

- **somCheckId**
- **somRegisterId**
- **somIdFromString**
- **somStringFromId**
- **somCompareIds**
- **somTotalRegIds**
- **somSetExpectedIds**

somUniqueKey

Example Code

```
#include <som.h>
main()
{
   unsigned long k1, k2;
   k1 = somUniqueKey(somIdFromString("abc"));
   k2 = somUniqueKey(somIdFromString("abc"));
   SOM_Test(k1 == k2);
}
```

somVaBuf_add

Adds an argument to the SOM buffer (**somVaBuf**) for variable arguments.

Syntax

```
long somVaBuf_add   (somVaBuf  vb, char  *arg, int  type)
```

Parameters

vb (somVaBuf)
Value (**somVaBuf**) returned from **somVaBuf_create** function.

arg (char *)
Pointer to the argument to be added to the **va_list**.

type (int)
Argument type (**TCKind**).

The following are the supported **TCKind** types:

- **tk_short**
- **tk_ushort**
- **tk_long**
- **tk_ulong**
- **tk_float**
- **tk_double**
- **tk_char**
- **tk_boolean**
- **tk_octet**
- **tk_Typecode**
- **tk_enum**
- **tk_string**
- **tk_pointer**

Returns

rc (long)

If successful, a value of one is returned; otherwise, a value of zero is returned.

somVaBuf_add

Remarks

This function adds the argument pointed to by *arg* to the **va_list** by using *type* for the size.

Related Information

Functions

- **somVaBuf_create**
- **somVaBuf_get_valist**
- **somVaBuf_destroy**
- **somvalistGetTarget**
- **somvalistSetTarget**

Example Code

```
See function somVaBuf_create.
```

somVaBuf_create

Creates a SOM buffer (**somVaBuf**) for variable arguments from which the **va_list** will be built.

Syntax

```
somVaBuf somVaBuf_create   (char  *vb, int  size)
```

Parameters

vb (char *)
Pointer to user-allocated memory or NULL.

size (int)
Size of memory pointed at by *vb*, or else zero.

Returns

rc (somVaBuf)

If successful, **somVaBuf** is returned; otherwise, a NULL value is returned.

Remarks

This function allocates, if necessary, and initializes a **somVaBuf** data structure. Memory is allocated if:

- The size parameter is less than the size of the **somVaBuf** structure,
- The size parameter is zero, or
- The *vb* parameter is NULL.

Note: Because the **somVaBuf** data structure is opaque, users cannot determine its size. Although this function accepts a user-allocated buffer, it is recommended that a NULL value be passed as the first argument.

Related Information

Functions

- **somVaBuf_add**
- **somVaBuf_get_valist**
- **somVaBuf_destroy**

- **somvalistGetTarget**
- **somvalistSetTarget**

Example Code

C Example

```
#include <somobj.h>

void f1(SOMObject obj, Environment *ev)
{
        char *msg;
        va_list start_val;
        somVaBuf vb;
        char *msg1 = "Good Morning";

        vb = (somVaBuf)somVaBuf_create(NULL, 0);
        somVaBuf_add(vb, (char *)&obj, tk_pointer);
                                        /* target for _set_msg */
        somVaBuf_add(vb, (char *)&ev, tk_pointer);
                                          /* next argument */
        somVaBuf_add(vb, (char *)&msg1, tk_pointer);
                                         /* final argument */
        somVaBuf_get_valist(vb, &start_val);

        /* dispatch _set_msg on object */
        SOMObject_somDispatch(
                obj,       /* target for somDispatch */
                0,         /* says ignore dispatched method result */
                somIdFromString("_set_msg"), /* the somId for _set_msg */
                start_val); /* target and args for _set_msg */

        /* dispatch _get_msg on obj:   */
        /* Get a fresh copy of the va_list    */
        somVaBuf_get_valist(vb, &start_val);
        SOMObject_somDispatch(
                obj,
                (somToken *)&msg,
                           /* address to store dispatched result */
                somIdFromString("_get_msg"),
                start_val); /* target and arguments for _get_msg */
        printf("%s\n", msg);
        somVaBuf_destroy(vb);
}
```

C++ Example

```
#include <somobj.h>

void f1(SOMObject obj, Environment *ev)
{
        char *msg;
        va_list start_val;
        somVaBuf vb;
        char *msg1 = "Good Morning";

        vb = (somVaBuf)somVaBuf_create(NULL, 0);
        somVaBuf_add(vb, (char *)&obj, tk_pointer);
                                                /* target for _set_msg */
        somVaBuf_add(vb, (char *)&ev, tk_pointer);
                                                 /* next argument */
        somVaBuf_add(vb, (char *)&msg1, tk_pointer);
                                                /* final argument */
        somVaBuf_get_valist(vb, &start_val);

        /* dispatch _set_msg on obj: */
        obj->SOMObject_somDispatch(
                0,         /* says ignore the dispatched method result */
                somIdFromString("_set_msg"), /* the somId for _set_msg */
                start_val);   /* the target and arguments for _set_msg */

        /* dispatch _get_msg on obj:   */
        /* Get a fresh copy of the va_list    */
        somVaBuf_get_valist(vb, &start_val);
        obj->SOMObject_somDispatch(
                (somToken *)&msg,
                           /* address to hold dispatched method result */
                somIdFromString("_get_msg"),
                start_val);   /* the target and arguments for _get_msg */
        printf("%s\n", msg);
        somVaBuf_destroy(vb);
}
```

somVaBuf_destroy

Releases the SOM buffer (**somVaBuf**) and its associated **va_list**.

Syntax

```
void somVaBuf_destroy   (somVaBuf  vb)
```

Parameters

vb (somVaBuf)
Value (**somVaBuf**) returned from **somVaBuf_create** function.

Returns

rc (void)

Remarks

If **somVaBuf** was allocated by the **somVaBuf_create** function, the memory will be deallocated.

Related Information

Functions

- **somVaBuf_create**
- **somVaBuf_add**
- **somVaBuf_get_valist**
- **somvalistGetTarget**
- **somvalistSetTarget**

Example Code

```
See function somVaBuf_create.
```

somVaBuf_get_valist

Initializes a **va_list** from the SOM buffer (**somVaBuf**).

Syntax

```
void somVaBuf_get_valist  (somVaBuf  vb, va_list  *ap)
```

Parameters

vb (somVaBuf)
Value (**somVaBuf**) returned from **somVaBuf_create** function.

ap (va_list *)
Pointer to a **va_list**.

Returns

rc (void)

None. The user's **va_list** has been initialized from the **va_list** in **somVaBuf**.

Remarks

This function copies the **va_list** in the **somVaBuf** structure to the passed **va_list**.

Related Information

Functions

- **somVaBuf_create**
- **somVaBuf_add**
- **somVaBuf_destroy**
- **somvalistGetTarget**
- **somvalistSetTarget**

Example Code

```
See function somVaBuf_create.
```

somvalistGetTarget

Gets the first scalar value from a **va_list** without other side effects.

Syntax

```
unsigned long somvalistGetTarget  (va_list  ap)
```

Parameters

ap (va_list)
The **va_list** from which to get the value.

Returns

rc (unsigned long)

Scalar value from the **va_list**.

Remarks

Returns the first scalar value from the **va_list** without other side effects.

Related Information

Functions

- **somVaBuf_create**
- **somVaBuf_add**
- **somVaBuf_get_valist**
- **somVaBuf_destroy**
- **somvalistSetTarget**

Example Code

```
va_list start_val;
somVaBuf vb;
unsigned long first;

vb = (somVaBuf)somVaBuf_create(NULL, 0);
...
first = somvalistGetTarget(start_val);
...
somvalistSetTarget(start_val, first);
```

somvalistSetTarget

Modifies the **va_list** without other side effects.

Syntax

```
unsigned long somvalistSetTarget  (va_list  ap, unsigned long  val)
```

Parameters

ap (va_list)
The **va_list** to modify.

val (unsigned long)
Value to set in the first scalar slot.

Returns

rc (unsigned long)

None.

Remarks

The **somvalistSetTarget** function replaces the first scalar value on the **va_list** with the value *val* that is passed in the call without any other side effects.

Related Information

Functions

- **somVaBuf_create**
- **somVaBuf_add**
- **somVaBuf_get_valist**
- **somVaBuf_destroy**
- **somvalistGetTarget**

Example Code

```
va_list start_val;
somVaBuf vb;
unsigned long first;

vb = (somVaBuf)somVaBuf_create(NULL, 0);
...
first = somvalistGetTarget(start_val);
...
somvalistSetTarget(start_val, first);
```

somVprintf

This function prints a formatted string in the manner of the C vprintf function.

Syntax

```
long somVprintf  (string  fmt, va_list  ap)
```

Parameters

fmt (string)
The format string to be output.

ap (va_list)
A **va_list** representing the values to be substituted into the format string.

Returns

rc (long)

Returns the number of characters written.

Remarks

This function prints a formatted string using SOMOutCharRoutine, in the same manner as the C vprintf function. The implementation of SOMOutCharRoutine determines the destination of the output, while the C printf function is always directed to stdout. (The default output destination for SOMOutCharRoutine is stdout also, but this can be modified by the user.)

Related Information

Data Structures

- **string (somcorba.h)**
- **va_list (stdarg.h)**

Functions

- **somPrintf**
- **somPrefixLevel**
- **somLPrintf**
- **SOMOutCharRoutine**

Example Code

```
#include <som.h>
main()
{
   va_list args;
   somVaBuf vb;
   float f = 3.1415;
   char c = 'a';
   int one = 1;
   char *msg = "This is a test";

  somEnvironmentNew();  /* Init environment */
   vb = (somVaBuf)somVaBuf_create(NULL, 0);
   somVaBuf_add(vb, (char *)&one, tk_long);
   somVaBuf_add(vb, (char *)&f, tk_float);
   somVaBuf_add(vb, (char *)&c, tk_char);
   somVaBuf_add(vb, (char *)&msg, tk_pointer);
   somVaBuf_get_valist(vb, &args);

   somVprintf("%d, %f, %c, %s\n", args);
}
```

SOMCalloc

This function allocates sufficient memory for an array of objects of a specified size.

Syntax

```
somToken (*SomCalloc)   (size_t  num, size_t  size)
```

Parameters

num (size_t)
The number of objects for which space is to be allocated.

size (size_t)
The size of the objects for which space to is to be allocated.

Returns

rc (somToken)

A pointer to the first byte of the allocated space.

Remarks

This function allocates an amount of memory equal to *num*size* (sufficient memory for an array of *num* objects of size *size).* This function has the same interface as the C **calloc** function. It performs the same basic function as **calloc** with some supplemental error checking. If an error occurs, the **SOMError** function is called. This routine is replaceable by changing the value of the global variable **SOMCalloc**.

Related Information

Data Structures

- **somToken (sombtype.h)**

Functions

- **SOMMalloc**
- **SOMRealloc**
- **SOMFree**

Example Code

```
See function somVprintf.
```

somClassInitFuncName

This function returns the name of the function used to initialize classes in a DLL.

Syntax

```
string (*somClassInitFuncName)  ()
```

Parameters

None.

Returns

rc (string)

Returns the name of the function that should be used to initialize classes in a DLL.

Remarks

This function is called by the SOM Class Manager to determine what function to call to initialize the classes in a DLL. The default version returns the string "SOMInitModule." The function can be replaced (so that the Class Manager will invoke a different function to initialize classes in a DLL) by changing the value of the global variable **SOMClassInitFuncName**.

Related Information

Data Structures

- **string (somcorba.h)**

Functions

- **SOMLoadModule**
- **SOMDeleteModule**

Example Code

```
#include <som.h>
string XYZFuncName() { return "XYZ"; }
main()
{
   SOMClassInitFuncName = XYZFuncName;
   ...
}
```

SOMDeleteModule

This function unloads a dynamically linked library (DLL).

Syntax

```
int (*SOMDeleteModule)  (somToken  modHandle)
```

Parameters

modHandle (somToken)
The **somToken** for the DLL to be unloaded. This token is supplied by the **SOMLoadModule** function when it loads the DLL.

Returns

rc (int)

0	Returns 0 if successful.
code	Returns a non-zero system-specific error code if not successful.

Remarks

This function unloads the specified dynamically linked library (DLL). This routine is called by the SOM Class Manager to unload DLLs. This function can be replaced (thus changing the way the Class Manager unloads DLLS) by changing the value of the global variable **SOMDeleteModule**.

Related Information

Data Structures

- **somToken (sombtype.h)**

Functions

- **SOMLoadModule**
- **SOMClassInitFuncName**

SOMError

This functions handles an error condition.

Syntax

```
void (*SOMError)  (int  errorCode, string  fileName, int  lineNum)
```

Parameters

errorCode (int)
An integer representing the error code of the error.

fileName (string)
The name of the file in which the error occurred.

lineNum (int)
The line number where the error occurred.

Returns

rc (void)

Remarks

This function inspects the specified error code and takes appropriate action, depending on the severity of the error. The last digit of the error code indicates whether the error is classified as SOM_Fatal (9), SOM_Warn (2), or SOM_Ignore (1). The default implementation of this function prints a message that includes the specified error code, filename, and line number, and terminates the current process if the error is classified as SOM_Fatal. The fileName and lineNum arguments specify where the error occurred. This routine can be replaced by changing the value of the global variable **SOMError**.

For C and C++ programmers, SOM defines a convenience macro, SOM_Error, which invokes this function and supplies the last two arguments.

SOMError

Related Information

Macros

- **SOM_Test**
- **SOM_TestC**
- **SOM_WarnMsg**
- **SOM_Assert**
- **SOM_Expect**
- **SOM_Error**

somFree

This function frees the specified block of memory.

Syntax

```
void (*somFree)  (somToken  ptr)
```

Parameters

ptr (somToken)
A pointer to the block of storage to be freed.

Returns

rc (void)

Remarks

This function frees the block of memory pointed to by *ptr.* This function should only be called with a pointer previously allocated by **SOMMalloc** or **SOMCalloc.** This function has the same interface as the C **free** function. It performs the same basic function as **free** with some supplemental error checking. If an error occurs, the **SOMError** function is called. This routine is replaceable by changing the value of the global variable **SOMFree.**

To free an *object* (rather than a block of memory), use the **somFree** method, rather than this function.

Related Information

Functions

- **SOMCalloc**
- **SOMRealloc**
- **SOMMalloc**

Methods

- **somFree**

Example Code

```
#include <som.h>
main()
{
   somToken ptr = SOMMalloc(20);
   SOMFree(ptr);
}
```

SOMInitModule

This function invokes the class creation routines for the classes contained in an OS/2 or Windows class library (DLL).

Syntax

```
void (*SOMInitModule)  (long  majorVersion, long  minorVersion,
                        string  className)
```

Parameters

majorVersion (long)
The major version number of the class that was requested when the library was loaded.

minorVersion (long)
The minor version number of the class that was requested when the library was loaded.

className (string)
The name of the class that was requested when the library was loaded.

Returns

rc (void)

Remarks

On OS/2 or Windows, a class library (DLL) can contain the implementations for multiple classes, all of which should be created when the DLL is loaded. On OS/2, when loading a DLL, the SOM class manager determines the name of a DLL initialization function, and if the DLL exports a function of this name, the class manager invokes that function (whose purpose is to create the classes in the DLL). **SOMInitModule** is the default name for this DLL initialization function.

On Windows, the SOM class manager does *not* call this function. It must be called from the default Windows DLL initialization function, LibMain. This call is made indirectly through the **SOM_ClassLibrary** macro (see the Example).

SOMInitModule

Related Information

Functions

- **SOMClassInitFuncName**

Methods

- **somGetInitFunction**

Macros

- **SOM_ClassLibrary**

Example Code

```
#include "xyz.h"
#ifdef __IBMC__
  #pragma linkage (SOMInitModule, system)
#endif

SOMEXTERN void  SOMLINK SOMInitModule (long majorVersion,
                          long minorVersion, string className)
{
    SOM_IgnoreWarning (majorVersion);  /* This function makes  */
    SOM_IgnoreWarning (minorVersion);  /* no use of the passed */
    SOM_IgnoreWarning (className);     /* arguments.   */
    xyzNewClass, (A_MajorVersion, A_MinorVersion);
}
```

For Windows, also include the following function:

```
#include <windows.h>
int CALLBACK LibMain (HINSTANCE inst,
                      WORD ds,
                      WORD Heapsize,
                      LPSTR cmdLine)
{
    SOM_IgnoreWarning (inst);
    SOM_ignoreWarning (ds);
    SOM_IgnoreWarning (heapSize);
    SOM_IgnoreWarning (cmdLine);

    SOM_ClassLibrary ("xyz.dll");
    return 1;  /* Indicate success to loader */
}
```

SOMLoadModule

This function loads the dynamically linked library (DLL) containing a SOM class.

Syntax

```
int (*SOMLoadModule)  (string  className, string  fileName,
                       string  functionName, int  majorVersion,
                       long  minorVersion, somToken  *modHandle)
```

Parameters

className (string)
The name of the class whose DLL is to be loaded.

fileName (string)
The name of the DLL library file. This can be either a simple name or a fully-qualified pathname.

functionName (string)
The name of the routine to be called after the DLL is loaded. The routine is responsible for creating a class object for each class in the DLL. Typically, this argument will have the value **SOMInitModule,** obtained from the **SOMClassInitFuncName** function. If no **SOMInitModule** entry exists in the DLL, the default version of this function looks for a routine named NewClass instead. If neither entry point is found, the default version of this function fails.

majorVersion (int)
The expected major version number of the class, to be passed to the initialization routine of the DLL.

minorVersion (long)
The expected minor version number of the class, to be passed to the initialization routine of the DLL.

modHandle (somToken *)
The address where this function should place a token that can be subsequently used by the **SOMDeleteModule** routine to unload the DLL.

Returns

rc (int)

0	Returns 0 if successful.
code	Returns a non-zero system-specific error code if not successful.

Remarks

This function loads the dynamically linked library (DLL) containing a SOM class. This routine is called by the SOM Class Manager to load DLLs. This function can be replaced (thus changing the way the Class Manager loads DLLS) by changing the value of the global variable **SOMLoadModule**.

Related Information

Functions

- **SOMDeleteModule**
- **SOMClassInitFuncName**

SOMMalloc

This function allocates the specified amount of memory.

Syntax

```
somToken (*SOMMalloc)   (size_t  size)
```

Parameters

size (size_t)
The amount of memory to be allocated, in bytes.

Returns

rc (somToken)
A pointer to the first byte of the allocated space.

Remarks

This function allocates *size* bytes of memory. This function has the same interface as the C **malloc** function. It performs the same basic function as **malloc** with some supplemental error checking. If an error occurs, the **SOMError** function is called. This routine is replaceable by changing the value of the global variable **SOMMalloc.**

Related Information

Functions

- **SOMCalloc**
- **SOMRealloc**
- **SOMFree**

Example Code

```
See function SOMFree.
```

SOMOutCharRoutine

This function prints a character. This function is replaceable.

Syntax

```
int (*SOMOutCharRoutine)  (char  c)
```

Parameters

c (char)
The character to be output.

Returns

rc (int)

0 Returns 0 if an error occurs.
1 Returns 1 if no error occurs.

Remarks

This function is a replaceable character output routine. It is invoked by SOM whenever a character is generated by one of the SOM error-handling or debugging macros. The default implementation outputs the specified character to stdout. To change the destination of character output, store the address of a user-written character output routine in global variable **SOMOutCharRoutine**.

Another function, **somSetOutChar**, may be preferred over the **SOMOutCharRoutine** function. The **somSetOutChar** function enables each application (or thread) to have a customized character output routine.

Related Information

Functions

- **somVprintf**
- **somPrefixLevel**
- **somLPrintf**
- **somPrinf**
- **somSetOutChar**

SOMOutCharRoutine

Example Code

```
#include <som.h>
#pragma linkage(myCharacterOutputRoutine, system)
/* Define a replacement routine: */
int SOMLINK myCharacterOutputRoutine (char c)
{
    (Customized code goes here)
}
...
/* After the next stmt all output */
/* will be sent to the new routine */
SOMOutCharRoutine = myCharacterOutputRoutine;
```

SOMRealloc

This function changes the size of a previously allocated region of memory.

Syntax

```
somToken (*SOMRealloc)  (somToken  ptr, size_t  size)
```

Parameters

ptr (somToken)
A pointer to the previously allocated region of memory. If NULL, a new region of memory of *size* bytes is allocated.

size (size_t)
The size in bytes for the re-allocated storage. If zero, the memory pointed to by *ptr* is freed.

Returns

rc (somToken)
A pointer to the first byte of the re-allocated space. (A pointer is returned because the block of storage may need to be moved to increase its size.)

Remarks

The **SOMRealloc** function changes the size of the previously allocated region of memory pointed to by *ptr* so that it contains *size* bytes. The new size may be greater or less than the original size. The **SOMRealloc** function has the same interface as the C **realloc** function. It performs the same basic function as **realloc** with some supplemental error checking. If an error occurs, the **SOMError** function is called. This routine is replaceable by changing the value of the global variable **SOMRealloc.**

Related Information

Functions:

- **SOMCalloc**
- **SOMMalloc**
- **SOMFree**

SOM_Assert

This macro asserts that a **boolean** condition is true.

Syntax

```
void SOM_Assert  (boolean  condition, long  errorCode)
```

Parameters

condition (boolean)
A **boolean** expression that is expected to be TRUE (nonzero).

errorCode (long)
The integer error code for the error to be raised if *condition* is FALSE.

Returns

rc (void)

Remarks

The **SOM_Assert** macro is used to place **boolean** assertions in a program:

- If *condition* is FALSE, and *errorCode* indicates a warning-level error and **SOM_WarnLevel** is set to be greater than zero, then a warning message is output.
- If *condition* is FALSE and *errorCode* indicates a fatal error, an error message is output and the process is terminated.
- If *condition* is TRUE and **SOM_AssertLevel** is set to be greater than zero, then an informational message is output.

External (Global) Data

```
long SOM_WarnLevel;   /* default = 0 */
long SOM_AssertLevel; /* default 0 */
```

Expansion

If *condition* is FALSE, and *errorCode* indicates a warning-level error and **SOM_WarnLevel** is set to be greater than zero, then a warning message is output. If *condition* is FALSE and *errorCode* indicates a fatal error, an error message is

output and the process is terminated. If *condition* is TRUE and **SOM_AssertLevel** is set to be greater than zero, then an information message is output.

Related Information

Macros:

- **SOM_Expect**
- **SOM_Test**
- **SOM_TestC**

Example Code

```
#include <som.h>
main()
{
   SOM_WarnLevel = 1;
   SOM_Assert(2==2, 29);
{
```

SOM_ClassLibrary

This macro identifies the file name of the DLL for a SOM class library in a Windows LibMain function.

Syntax

```
void SOM_ClassLibrary   (string  libname.dll)
```

Parameters

libname.dll (string)
The name of the file containing the DLL (as the name would appear in a Windows LoadLibrary call).

Returns

rc (void)
None.

Remarks

Each Windows SOM class library must supply a Windows LibMain function. In LibMain, this macro identifies both the actual file name of the library as it would appear in a Windows LoadLibrary call and the location of the library's **SOMInitModule** function. This information is passed to the SOM Kernel, which in turn registers the library and schedules the execution of the **SOMInitModule** function. This macro can also be used in OS/2 class libraries within the context of a DLL “init/term” function.

Typically, the SOM Kernel invokes the **SOMInitModule** function of each statically loaded class library during the execution of the **somMainProgram** function in the using application. For dynamically loaded class libraries, **SOMInitModule** is invoked immediately upon completion of the library's LibMain (or an OS/2 DLL “init/term”) function.

Because this macro expands to reference the **SOMInitModule** function, either a declaration of the **SOMInitModule** function, or the function itself, should precede the appearance of **SOM_ClassLibrary** in the current compilation unit, as shown in the Example below).

The Example illustrates the use of the **SOM_ClassLibrary** macro in a Windows LibMain function.

Related Information

Macros

- **SOM_MainProgram**

Functions

- **somMainProgram**

Example Code

```
#include <som.h>
SOMEXTERN void SOMLINK SOMInitModule (long majorVersion,
                                      long minorVersion,
                                      string className);

#include <windows.h>
int CALLBACK LibMain (HINSTANCE inst,
                      WORD ds,
                      WORD Heapsize,
                      LPSTR cmdLine)
{
    SOM_IgnoreWarning (inst);
    SOM_ignoreWarning (ds);
    SOM_IgnoreWarning (heapSize);
    SOM_IgnoreWarning (cmdLine);

    SOM_ClassLibrary ("xyz.dll");
    return 1;  /* Indicate success to loader */
}
```

SOM_CreateLocalEnvironment Macro

This macro creates and initializes a local **Environment** structure.

Syntax

```
Environment * SOM_CreateLocalEnvironment Macro ()
```

Parameters

None.

Returns

rc (Environment *)

Remarks

The **SOM_CreateLocalEnvironment** macro creates a local **Environment** structure. This **Environment** structure can be passed to methods as the **Environment** argument so that exception information can be returned without affecting the global environment.

Expansion

The **SOM_CreateLocalEnvironment** expands to an expression of type (**Environment ***).

Related Information

Data Structures:

- **Environment** (somcorba.h)

Macros:

- **SOM_DestroyLocalEnvironment**
- **SOM_InitEnvironment**
- **SOM_UninitEnvironment**

Functions:

- **somGetGlobalEnvironment**

Example Code

```
Environment *ev;
ev = SOM_CreateLocalEnvironment();
_myMethod(obj, ev);
...
SOM_DestroyLocalEnvironment(ev);
```

SOM_DestroyLocalEnvironment

This macro destroys a local **Environment** structure.

Syntax

```
void SOM_DestroyLocalEnvironment   (Environment  *env)
```

Parameters

env (Environment *)
A pointer to the **Environment** structure to be discarded.

Returns

rc (void)

Remarks

The **SOM_DestroyLocalEnvironment** macro destroys a local **Environment** structure, such as one created using the **SOM_CreateLocalEnvironment** macro.

Related Information

Macros:

- **SOM_CreateLocalEnvironment**
- **SOM_UninitEnvironment**

Functions:

- **somExceptionFree**

Example Code

```
Environment *ev;
ev = SOM_CreateLocalEnvironment();
_myMethod(obj, ev);
...
SOM_DestroyLocalEnvironment(ev);
```

SOM_Error

This macro reports an error condition.

Syntax

```
void SOM_Error  (long  errorCode)
```

Parameters

errorCode (long)
The integer error code for the error to be reported.

Returns

void (void)

Remarks

The **SOM_Error** macro invokes the **SOMError** error handling procedure with the specified error code, supplying the filename and line number where the macro was invoked. The default implementation of **SOMError** outputs a message containing the error code, filename, and line number. Additionally, if the last digit of the error code indicates a serious error (that is, value SOM_Fatal), the process is terminated.

Related Information

Functions:

- **SOMError**

SOM_Expect

This macro asserts that a **boolean** condition is expected to be true.

Syntax

```
void SOM_Expect   (boolean  condition)
```

Parameters

condition (boolean)
A boolean expression that is expected to be TRUE (nonzero).

Returns

rc (void)

Remarks

The **SOM_Expect** macro is used to place **boolean** assertions that are expected to be true into a program:

- If *condition* is FALSE 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.

Expansion

If *condition* is FALSE 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 information message is output.

Related Information

Macros:

- **SOM_Assert**
- **SOM_Test**
- **SOM_TestC**

SOM_Expect

Example Code

```
SOM_Expect(2==2);
```

SOM_GetClass

This macro returns the class object of which a SOM object is an instance.

Syntax

```
SOMClass SOM_GetClass   (SOMObject  objPtr)
```

Parameters

objPtr (SOMObject)
A pointer to the object whose class is needed.

Returns

rc (SOMClass)

Remarks

The **SOM_GetClass** macro returns the class object of which *obj* is an instance. This is done without recourse to a method call on the object. The **somGetClass** method introduced by SOMObject is also intended to return the class of which an object is an instance, and the default implementation provided for this method by SOMObject uses the macro.

Important Note: It is generally recommended that the **somGetClass** method call be used, since it cannot be known whether the class of an object wishes to provide special handling when its address is requested from an instance. But, there are (rare) situations where a method call cannot be made, and this macro can then be used. If you are unsure as to whether to use the method or the macro, you should use the method.

Related Information

Methods:

- **somGetClass**

Example Code

```
#include <somcls.xh>
#include <animal.xh>
main()
{
   Animal *a = new Animal;
   SOMClass cls1 = SOM_GetClass(a);
   SOMClass cls2 = a->somGetClass();
   if (cls1 == cls2)
      printf("macro and method for getClass the same for Animal\n");
   else
      printf("macro and method for getClass not same for Animal\n");
}
```

SOM_InitEnvironment

This macro initializes a local **Environment** structure.

Syntax

```
void SOM_InitEnvironment  (Environment  *env)
```

Parameters

env (Environment *)
A pointer to the **Environment** structure to be initialized.

Returns

rc (void)

Remarks

The **SOM_InitEnvironment** macro initializes a locally declared **Environment** structure. This **Environment** structure can then be passed to methods as the **Environment** argument so that exception information can be returned without affecting the global environment.

Expansion

The **SOM_InitEnvironment** initializes an **Environment** structure to zero.

Related Information

Macros:

- **SOM_DestroyLocalEnvironment**
- **SOM_CreateLocalEnvironment**
- **SOM_UninitEnvironment**

Functions:

- **somGetGlobalEnvironment**

SOM_InitEnvironment

Example Code

```
Environment ev;
SOM_InitEnvironment(&ev);
_myMethod(obj, &ev);
...
SOM_UninitEnvironment(&ev);
```

SOM_MainProgram

This macro identifies an application as a SOM program and registers an end-of-program exit procedure to release SOM resources when the application terminates.

Syntax

```
SOM_MainProgram ()
```

Parameters

None

Remarks

This macro should appear near the beginning of each Windows application program that uses SOM or a SOM class library. It can also be used in OS/2 or AIX programs but is not generally required on these platforms. Any statically referenced SOM class libraries are initialized during the execution of this macro, and an end-of-program exit procedure is established to release SOM resources during normal program termination. (This macro combines the execution of the C/C++ "atexit" function with the SOM **somMainProgram** function and returns a reference to the global **SOMClassMgr** object.)

Related Information

Functions

- **somMainProgram**

Macros

- **SOM_ClassLibrary**

SOM_MainProgram

Example Code

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

int PASCAL WinMain (HINSTANCE inst,
                    WORD ds,
                    WORD Heapsize,
                    LPSTR cmdLine)
{
    ...
    SOM_MainProgram ();
    ...
    /* Rest of main program follows */
}
```

SOM_NoTrace

This macro is used to turn off method debugging.

Syntax

```
SOM_NoTrace  (<token> className, <token> methodName)
```

Parameters

className (<token>)
The name of the class for which tracing will be turned off, given as a simple token rather than as a quoted string.

methodName (<token>)
The name of the method for which tracing will be turned off, given as a simple token rather than as a quoted string.

Remarks

The **SOM_NoTrace** macro is used to turn off method debugging. Within an implementation file for a class, before #including the implementation (.ih or .xih) header file for the class, #define the **<className>MethodDebug** macro to be **SOM_NoTrace**. Then, **<className>MethodDebug** will have no effect.

Expansion

The **SOM_NoTrace** macro has a null (empty) expansion.

Example Code

```
#define AnimalMethodDebug(c,m) SOM_NoTrace(c,m)
#include <animal.ih>
/* Now AnimalMethodDebug does nothing */
```

SOM_ParentNumResolve

This macro obtains a pointer to a method procedure from a list of method tables. It is used by C and C++ implementation bindings to implement parent method calls.

Syntax

```
SOM_ParentNumResolve  ((<token> introClass, long parentNum,
                       somMethodTab parentMtabs, <token>
                       ) methodName)
```

Parameters

introClass ((<token>)
The name of the class that introduces *methodName*. This name should be given as a simple token, rather than a quoted string (for example, *Animal*. rather than "*Animal*").

parentNum (long)
The position of the desired parent. The first (leftmost) parent of a class has position 1.

parentMtabs (somMethodTab)
A list of parent method tables acquired by invoking the **somGetPClsMtabs** method on a class object.

methodName (<token>))
The name of the method to be resolved. This name should be given as a simple token, rather than a quoted string (for example, *setSound* rather than "*setSound*").

Remarks

This macro invokes the **somParentNumResolve** function to obtain a pointer to the static method procedure that implements the specified method for the specified parent. The method is specified by indicating the introducing class, *IntroClass*, and the method name, *methodName*.

Expansion

The expansion of the macro produces an expression that is appropriately typed for application of the evaluated result to the indicated method's arguments, as illustrated in the Example.

Related Information

Functions

- **somParentNumResolve**

Methods

- **somGetPClsMtabs**

Example Code

```
#include <somcls.h>

main()
{
 SOMClassMgr * cm = somEnvironmentNew();
 somMethodTabs mList = _somGetPClsMtabs(_SOMClass);
 SOM_ParentNumResolve(SOMObject, 1, mList, somDumpSelfInt)
      (_SOMClass,1);
}
```

SOM_Resolve

This macro obtains a pointer to a method procedure.

Note: For backward compatibility, this method does *not* take an **Environment** parameter.

Syntax

```
RC SOM_Resolve  (SOMObject  objPtr, <token>  className,
                 <token>  methodName)
```

Parameters

objPtr (SOMObject)
A pointer to the object to which the resolved method procedure will be applied.

className (<token>)
The name of the class that introduces *methodName*. This name should be given as a simple token, rather than a quoted string (for example, *Animal* rather than "*Animal*").

methodName (<token>)
The name of the method to be resolved. This name should be given as a simple token, rather than a quoted string (for example, *setSound* rather than "*setSound*").

Returns

somMethod (RC)

Remarks

The **SOM_Resolve** macro invokes the **somResolve** function to obtain a pointer to the static method procedure that implements the specified method for the specified object. This pointer can be used for efficient repeated casted method invocations on instances of the class of the object on which the resolution is done, or instances of subclasses of this class. The name of the class that introduces the method and the name of the method must be known to use this macro. Otherwise, use the **somResolveByName, somFindMethod** or **somFindMethodOk** method.

The **SOM_Resolve** macro can only be used to obtain a method procedure for a static method (one defined in the IDL specification for a class); not a dynamic method. Unlike the **SOM_ResolveNoCheck** macro, the **SOM_Resolve** macro performs several consistency checks on the object pointed to by *objPtr.*

Expansion

The **SOM_Resolve** macro uses the *className* and *methodName* to construct the method token for the specified method, then invokes the **somResolve** function. Thus, the macro expands to an expression that represents the entry-point address of the method procedure. This value can be stored in a variable and used for subsequent invocations of the method.

Related Information

Macros:

- **SOM_ResolveNoCheck**

Functions:

- **somResolve**
- **somClassResolve**
- **somResolveByName**

Methods:

- **somFindMethod**
- **somFindMethodOk**
- **somDispatch**
- **somClassDispatch**

Example Code

```
Animal myObj = AnimalNew();
somMethodProc *procPtr;
procPtr = SOM_Resolve(myObj, Animal, setSound);
/* note that procPtr will need to be typecast when it is used */
```

SOM_ResolveNoCheck

This macro obtains a pointer to a method procedure, without doing consistency checks.

Syntax

```
somMethodPtr SOM_ResolveNoCheck  (SOMObject  objPtr,
                                  <token>  className,
                                  <token>  methodName)
```

Parameters

objPtr (SOMObject)
A pointer to the object to which the resolved method procedure will be applied.

className (<token>)
The name of the class that introduces *methodName*. This name should be given as a simple token, rather than a quoted string (for example, *Animal* rather than "*Animal*").

methodName (<token>)
The name of the method to be resolved. This name should be given as a simple token, rather than a quoted string (for example, *setSound* rather than "*setSound*").

Returns

rc (somMethodPtr)

Remarks

The **SOM_ResolveNoCheck** macro invokes the **somResolve** function to obtain a pointer to the method procedure that implements the specified method for the specified object. This pointer can be used for efficient repeated invocations of the same method on the same type of objects. The name of the class that introduces the method and the name of the method must be known at compile time. Otherwise, use the **somFindMethod** or **somFindMethodOk** method.

The **SOM_ResolveNoCheck** macro can only be used to obtain a method procedure for a static method (one defined in the IDL specification for a class) and not a

method added to a class at run time. Unlike the **SOM_Resolve** macro, the **SOM_ResolveNoCheck** macro does not perform any consistency checks on the object pointed to by *objPtr*.

Expansion

The **SOM_ResolveNoCheck** macro uses the *className* and *methodName* to construct an expression whose value is the method token for the specified method, then invokes the **somResolve** function. Thus, the macro expands to an expression that represents the entry-point address of the method procedure. This value can be stored in a variable and used for subsequent invocations of the method.

Related Information

Macros:

- **SOM_Resolve**

Functions:

- **somResolve**
- **somClassResolve**
- **somResolveByName**

Methods:

- **somDispatch**
- **somClassDispatch**
- **somFindMethod**
- **somFindMethodOk**

Example Code

```
Animal myObj = AnimalNew();
somMethodProc *procPtr;
procPtr = SOM_ResolveNoCheck(myObj, Animal, setSound)
```

SOM_SubstituteClass

This macro provides a convenience macro for invoking the **somSubstituteClass** method.

Syntax

```
SOM_SubstituteClass  (<token> oldClass, <token> newClass)
```

Parameters

oldClass (<token>)
The name of the class to be substituted, given as a simple token rather than a quoted string.

newClass (<token>)
The name of the class that will replace *oldClass*, given as a simple token rather than a quoted string.

Remarks

The method **somSubstituteClass** requires existing class objects as arguments. Therefore, this macro first assures that the classes named *oldClass* and **newClass** exist, and then calls the method **somSubstituteClass** with these class objects as arguments.

Related Information

Methods

- **somSubstituteClass**

Example Code

```
See the method somSubstituteClass.
```

SOM_Test

This macro tests whether a **boolean** condition is true; if not, a fatal error is raised.

Syntax

```
void SOM_Test  (boolean  expression)
```

Parameters

expression (boolean)
The **boolean** expression to test.

Returns

rc (void)

Remarks

The **SOM_Test** macro tests the specified **boolean** expression:

- If the expression is TRUE and **SOM_AssertLevel** is set to a value greater than zero, then an information message is output.
- If the expression is FALSE, an error message is output and the process is terminated.

  Note: The **SOM_TestC** macro is similar, except that it only outputs a warning message in this situation.

External (Global) Data

```
long SOM_AssertLevel;  /* default is 0 */
```

Expansion

The **SOM_Test** macro tests the specified boolean expression. If the expression is TRUE and **SOM_AssertLevel** is set to a value greater than zero, then an information message is output. If the expression is FALSE, an error message is output and the process is terminated.

SOM_Test

Related Information

Macros:

- **SOM_Expect**
- **SOM_Assert**
- **SOM_TestC**

Example Code

```
#include <som.h>
main()
{
   SOM_AssertLevel = 1;
   SOM_Test(1=1);
}
```

SOM_TestC

This macro tests whether a **boolean** condition is true; if not, a warning message is output.

Syntax

```
void SOM_TestC  (boolean  expression)
```

Parameters

expression (boolean)
The **boolean** expression to test.

Returns

rc (void)

Remarks

The **SOM_TestC** macro tests the specified **boolean** expression:

- If the expression is TRUE and **SOM_AssertLevel** is set to a value greater than zero, then an information message is output.
- If the expression is FALSE and **SOM_WarnLevel** is set to a value greater than zero, then a warning message is output.

  Note: The **SOM_Test** macro is similar, except that it raises a fatal error in this situation.

External (Global) Data

```
long SOM_AssertLevel;  /* default is 0 */
long SOM_WarnLevel;    /* default is 0 */
```

Expansion

The **SOM_TestC** macro tests the specified **boolean** expression. If the expression is TRUE and **SOM_AssertLevel** is set to a value greater than zero, then an information message is output. If the expression is FALSE and **SOM_WarnLevel** is set to a value greater than zero, a warning message is output.

SOM_TestC

Related Information

Macros:

- **SOM_Expect**
- **SOM_Assert**
- **SOM_Test**

Example Code

```
#include <som.h>
main()
{
   SOM_WarnLevel = 1;
   SOM_TestC(1=1);
}
```

SOM_UninitEnvironment

This macro uninitializes a local **Environment** structure.

Syntax

```
void SOM_UninitEnvironment   (Environment  *env)
```

Parameters

env (Environment *)

Returns

rc (void)

Remarks

The **SOM_UninitEnvironment** macro uninitializes a locally declared **Environment** structure.

Expansion

The **SOM_UninitEnvironment** invokes the **somExceptionFree** function on the specified **Environment** structure.

Related Information

Macros:

- **SOM_DestroyLocalEnvironment**
- **SOM_InitEnvironment**

SOM_UninitEnvironment

Example Code

```
Environment ev;
SOM_InitEnvironment(&ev);
_myMethod(obj, &ev);
...
SOM_UninitEnvironment(&ev);
```

SOM_WarnMsg

This macro reports a warning message.

Syntax

```
void SOM_WarnMsg  (string  msg)
```

Parameters

msg (string)
The warning message to be output.

Returns

rc (void)

Remarks

If **SOM_WarnLevel** is set to a value greater than zero, the **SOM_WarnMsg** macro prints the specified message, along with the filename and line number where the macro was invoked.

Expansion

If **SOM_WarnLevel** is set to a value greater than zero, the **SOM_WarnMsg** macro prints the specified message, along with the filename and line number where the macro was invoked.

Related Information

Methods:

- **SOM_Error**

SOMClass

File stem: somcls

Base

SOMObject

Metaclass

SOMClass
(SOMClass is the only class with itself as metaclass.)

Ancestor Classes

SOMObject

Description

SOMClass is the root class for all SOM metaclasses. That is, all SOM metaclasses must be subclasses of **SOMClass** or some other class derived from it. It defines the essential behavior common to all SOM classes. In particular, it provides a suite of methods for initializing class objects, generic methods for manufacturing instances of those classes, and methods that dynamically obtain or update information about a class and its methods at run time.

Just as all SOM classes are expected to have **SOMObject** (or a class derived from **SOMObject**) as their base class, all SOM classes are expected to have **SOMClass** or a class derived from **SOMClass** as their metaclass. Metaclasses define "class" methods (sometimes called "factory" methods or "constructors") that manufacture objects from any class object that is defined as an instance of the metaclass.

To define your own class methods, define your own metaclass by subclassing **SOMClass** or one of its subclasses. Three methods that **SOMClass** inherits and overrides from **SOMObject** are typically overridden by any metaclass that introduces instance data **somInit**, **somUninit**, and **somDumpSelfInt**. The new methods introduced in **SOMClass** that are frequently overridden are **somNew**, **somRenew**, and **somClassReady**. (See the descriptions of these methods for further information.)

Other reasons for creating a new metaclass include tracking object instances, automatic garbage collection, interfacing to a persistent object store, or providing/managing information that is global to a set of object instances.

Types

```
typedef sequence <SOMClass> SOMClassSequence;

struct somOffsetInfo  {
   SOMClass          cls;
   long              offset
   };
typedef sequence <somOffsetInfo> SOMOffsets;
```

New Methods

Attributes:

```
readonly attribute somOffsets somInstanceDataOffsets
```

_get_somInstanceDataOffsets returns a sequence of structures, each of which indicates an ancestor of the receiver class (or the receiver class itself) and the offset to the beginning of the instance data introduced by the indicated class in an instance of the receiver class. The somOffsets information can be used in conjunction with information derived from calls to a *SOM Interface Repository* to completely determine the layout of SOM objects at runtime.

C++ Example

```
#include <somcls.xh>
main()
{
   int i;
   SOMClassMgr *scm = somEnvironmentNew();
   somOffsets so = _SOMClass->_get_somInstanceDataOffsets();
   for (i=0; i
      printf("In an instance of SOMClass, %s data starts at %d\n",
                so._bufferffi"->cls->somGetName(),
                so._bufferffi"->offset);
}
```

Introduced Methods

The following list shows all the SOMClass introduced methods.

Group: Instance Creation (Factory)

- somAllocate
- somDeallocate
- somNew
- somNewNoInit
- somRenew
- somRenewNoInit
- somRenewNoInitNoZero
- somRenewNoZero

Group: Initialization/Termination

- somAddDynamicMethod
- somClassReady

Group: Access

- somGetInstancePartSize
- somGetInstanceSize
- somGetInstanceToken
- somGetMemberToken
- somGetMethodData,
- somGetMethodDescriptor
- somGetMethodIndex
- somGetMethodToken
- somGetName
- somGetNthMethodData
- somGetNthMethodInfo
- somGetNumMethods
- somGetNumStaticMethods
- somGetParents
- somGetVersionNumbers

Group: Testing

- somCheckVersion
- somDescendedFrom
- somSupportsMethod

Group: Dynamic

- somFindMethod
- somFindMethodOk
- somFindSMethod
- somFindSMethodId
- somFindSMethodOk
- somLookupmethod

Overridden Methods

The following list shows all the methods overridden by the SOMClass class. These methods are overridden in order to modify the behavior defined by an ancestor class.

- somDumpSelfInt
- somDefaultInit
- somDestruct

Deprecated Methods

Use of the methods listed below is discouraged. There are three reasons for this:

First, these methods are used in constructing classes, and this capability is provided by the function **somBuildClass**. Class construction in SOM is currently a fairly complex activity, and it is likely to become even more so as the SOMobjects kernel evolves. To avoid breaking source code that constructs classes, you are advised to always use **somBuildClass** to build SOM classes. Note that the SOM language bindings always use **somBuildClass**.

Second, these methods are used for customizing aspects of SOM classes, such as method resolution and object creation. Doing this requires that metaclasses override various methods introduced by **SOMClass**. However, if this is done without the Cooperation Framework that implements the SOM Metaclass Framework, SOMobjects cannot guarantee that applications will function correctly. Unfortunately, the Cooperation Framework (while available to SOM users as an experimental feature) is not officially supported by the SOMobjects Toolkit. So, this is another reason why the following methods are deprecated.

Finally, some of these methods are now obsolete, so it seems appropriate that their use be discouraged.

- somAddStaticMethod
- somGetApplyStub
- somGetClassData
- somGetClassMtab
- somGetInstanceOffset
- somGetMethodOffset
- somGetParent
- somGetPCIsMtab
- somGetPCIsMtabs
- somGetRdStub
- somInitClass
- somInitMIClass
- somOverrideMtab
- somOverrideSMethod
- somSetClassData
- somSetMethodDescriptor
- _get_somDirectInitClasses attribute
- _set_somDirectInitClasses attribute

somAddDynamicMethod

This method adds a new dynamic instance method to a class. Dynamic methods are not part of the declared interface to a class of objects, and are therefore not supported by implementation and usage bindings. Instead, dynamic methods provide a way to dynamically add new methods to a class of objects during execution. SOM provides no standard protocol for informing a user of the existence of dynamic methods and the arguments they take. Dynamic methods must be invoked using name-lookup or dispatch resolution.

Note: For backward compatibility, this method does *not* take an **Environment** parameter.

Syntax

```
void somAddDynamicMethod  (SOMClass  receiver, somId  methodId,
                           somId  methodDescriptor,
                           somMethodPtr  method,
                           somMethodPtr  applyStub)
```

Parameters

receiver (SOMClass)
A pointer to a SOM class object.

methodId (somId)
A **somId** that names the method.

methodDescriptor (somId)
A **somId** appropriate for requesting information concerning the method from the SOM IR. This is currently of the form <className>: :<methodName>.

method (somMethodPtr)
A pointer to the procedure that will implement the new method. The first argument of this procedure is the address of the object on which it is being invoked.

applyStub (somMethodPtr)
A pointer to a procedure that returns nothing and receives as arguments: a method receiver; an address where the return value from the method call is to be stored; a pointer to a method procedure; and a va_list containing the arguments to the method. The applyStub procedure (which is usually called by **somDispatch**

must unload its va_list argument into separate variables of the correct type for the method, invoke its procedure argument on these variables, and then copy the result of the procedure invocation to the address specified by the return value argument.

Returns

rc (void)

Remarks

The **somAddDynamicMethod** method adds a new dynamic instance method to the receiving class. This involves recording the method's ID, descriptor, method procedure (specified by *method*), and apply stub in the receiving class's method data.

The arguments to this method should be non-null and obey the requirements expressed below. This is the responsibility of the implementor of a class, who in general has no knowledge of whether clients of this class will require use of the *applyStub* argument.

Original Class

SOMClass

Related Methods

Methods

- **somGetMethodDescriptor**

Example Code

```
/* New dynamic method "newMethod1" for class "XXX" */
static char *somMN_newMethod1 = "newMethod1";
static somId somId_newMethod1 = &somMN_newMethod1;
static char *somDS_newMethod1 = odq.XXX::newMethod1";
static somId somDI_newMethod1 = &somDS_newMethod1;

static void SOMLINK somAP_newMethod1(SOMObject somSelf,
                                     void *__retVal,
                                     somMethodProc *__methodPtr,
                                     va_list __ap)
{
void* __somSelf = va_arg(__ap, SOMObject);

int arg1 = va_arg(__ap, int);
SOM_IgnoreWarning(__retVal);
((somTD_SOMObject_newMethod1) __methodPtr) (__somSelf, arg1);
}
main()
{
_somAddDynamicMethod (
   XXXClassData.classObject,           /* Receiver (class object) */
   somId_newMethod1,                   /* method name somId */
   somDI_newMethod1,                   /* method descriptor somId */
   (somMethodProc%rbl.*) newMethod1,       /* method procedure */
   (somMethodProc *) somAP&newMethod1); /* method apply stub */
}
```

somAllocate

This method supports class-specific memory allocation for class instances. It cannot be overridden.

For backward compatibility, this method does *not* take an **Environment** parameter.

Syntax

```
string somAllocate   (SOMClass  receiver, long  size)
```

Parameters

receiver (SOMClass)
A pointer to the class object whose memory allocation method is desired.

size (long)
The number of bytes to be allocated.

Returns

rc (string)

string	A pointer to the first byte of the allocated memory region.
NULL	If sufficient memory is not available.

Remarks

When building a class, the **somBuildClass** function is responsible for registering the procedure that will be executed when this method is invoked on the class. The default procedure registered by **somBuildClass** uses the **SOMMalloc** function, but the IDL modifier **somallocate** can be used in the SOM IDL class implementation section to indicate a different procedure. Users of this method should be sure to use the dual method, **somDeallocate**, to free allocated storage. Also, if the IDL modifier **somallocate** is used to indicate a special allocation routine, the IDL modifier **somdeallocate** should be used to indicate a dual procedure to be called when the **somDeallocate** method is invoked.

somAllocate

Original Class

SOMClass

Related Methods

Methods

- **somDeallocate**

Example Code

```
#include <som.xh>
#<somcls.xh>
main()
{
   SOMClassMgr *cm = somEnvironmentNew();
   /* Use SOMClass's instance allocation method */
   string newRegion =  _SOMClass->somAllocate(20);
}
```

somCheckVersion

This method checks a class for compatibility with the specified major and minor version numbers. Not generally overridden.

Note: For backward compatibility, this method does *not* take an **Environment** parameter.

Syntax

```
boolean somCheckVersion  (SOMClass  receiver, long  majorVersion,
                          long  minorVersion)
```

Parameters

receiver (SOMClass)
A pointer to the SOM class whose version information should be checked.

majorVersion (long)
This value usually changes only when a significant enhancement or incompatible change is made to a class.

minorVersion (long)
This value changes whenever minor enhancements or fixes are made to a class. Class implementors usually maintain downward compatibility across changes in the *minorVersion* number.

Returns

rc (boolean)

1 Returns 1 (true) if the implementation of this class is compatible with the specified major and minor version number.
0 Returns 0 (false) if the implementation of this class is not compatible with the specified major and minor version number.

Remarks

This method checks the receiving class for compatibility with the specified major and minor version numbers. An implementation is compatible with the specified version numbers if it has the same major version number and a minor version number that is

equal to or greater than *minorVersion*. The version number pair (0,0) is considered to match any version.

This method is called automatically after creating a class object to verify that a dynamically loaded class definition is compatible with a client application.

Original Class

SOMClass

Example Code

```
#include <animal.h>
main()
{
   Animal myAnimal;
   myAnimal = AnimalNew();

   if (_somCheckVersion(_Animal, 0, 0))
      somPrintf("Animal IS compatible with 0.0\n");
   else
      somPrintf("Animal IS NOT compatible with 0.0\n");

   if (_somCheckVersion(_Animal, 1, 1))
      somPrintf("Animal IS compatible with 1.1\n");
   else
      somPrintf("Animal IS NOT compatible with 1.1\n";

   _somFree(myAnimal);
   }
```

Assuming that the implementation of Animal is version 1.0, this program produces the following output:

```
Animal IS compatible with 0.0
Animal IS NOT compatible with 1.1
```

somClassReady

This method indicates that a class has been constructed and is ready for normal use. Designed to be overridden.

For backward compatibility, this method does *not* take an **Environment** parameter.

Syntax

```
void somClassReady   (SOMClass  receiver)
```

Parameters

receiver (SOMClass)
A pointer to the class object that should be registered.

Returns

rc (void)

Remarks

This method is invoked automatically by the **somBuildClass** function after constructing and initializing a class object. The default implementation of this method provided by SOMClass simply registers the newly constructed class with **SOMClassMgrObject**. Metaclasses can override this method to augment class construction with additional registration protocol.

To have special processing done when a class object is created, you must define a metaclass for the class that overrides this method. The final statement in any overriding method should invoke the parent method to ensure that the class is properly registered with **SOMClassMgrObject**. Users of the C and C++ implementation bindings for SOM classes should never invoke this method directly; it is invoked automatically during class construction.

Original Class

SOMClass

somDeallocate

This method frees memory originally allocated by the **somAllocate** method from the same class object. It cannot be overridden.

For backward compatibility, this method does *not* take an **Environment** parameter.

Syntax

```
void somDeallocate  (SOMClass  receiver, string  memPtr)
```

Parameters

receiver (SOMClass)
A pointer to the class object whose **somAllocate** was originally used to allocate the memory now to be freed.

memPtr (string)
A pointer to the first byte of the region of memory that is to be freed.

Returns

rc (void)

Remarks

The **somDeallocate** method is intended for use to free memory allocated using its dual method, **somAllocate**. When building a class, the **somBuildClass** function is responsible for registering the procedure that will be executed when this method is invoked on the class. The default procedure regisistered by **somBuildClass** uses the **SOMFree** function, but the IDL modifier **somdeallocate** can be used in the SOM IDL class implementation section to indicate a different procedure. Users of this method should be sure that the dual method, **somAllocate**, was originally used to allocate storage. Also, if the IDL modifier **somdeallocate** is used to indicate a special deallocation routine, the IDL modifier **somallocate** should be used to indicate a dual procedure to be called when **somAllocate** is invoked.

Original Class

SOMClass

Related Methods

Methods

- **somAllocate**

somDescendedFrom

This method tests whether one class is derived from another. Not generally overridden.

For backward compatibility, this method does *not* take an **Environment** parameter.

Syntax

```
boolean somDescendedFrom   (SOMClass  receiver, SOMClass  aClassObj)
```

Parameters

receiver (SOMClass)
A pointer to the class object to be tested.

aClassObj (SOMClass)
A pointer to the potential ancestor class.

Returns

rc (boolean)

1 Returns 1 (true) if *receiver* is derived from *aClassObj*.
0 Returns 0 (false) if *receiver* is not derived from *aClassObj*.

Remarks

This method tests whether the receiver class is derived from a given class. For programs that use classes as types, this method can be used to ascertain whether the type of one object is a subtype of another.

This method considers a class object to be descended from itself.

Original Class

SOMClass

Related Methods

Methods

- **somIsA**
- **somIsInstanceOf**

Example Code

```
#include <dog.h>
/* ------------------------------------------
   Note: Dog is a subclass of Animal.
   ------------------------------------------ */
main()
{
  AnimalNewClass(0,0);
  DogNewClass(0,0);

  if (_somDescendedFrom (_Dog, _Animal))
     somPrintf("Dog IS descended from Animal\n");
  else
      somPrintf("Dog is NOT descended from Animal\n");
  if (_somDescendedFrom (_Animal, _Dog))
     somPrintf("Animal IS descended from Dog\n");
  else
     somPrintf("Animal is NOT descended from Dog\n");
```

This program produces the following output:

```
Dog IS descended from Animal
Animal is NOT descended from Dog
```

somFindMethod

This method finds the method procedure for a method and indicates whether it represents a static method or a dynamic method. Not generally overridden.

For backward compatibility, this method does not take an **Environment** parameter.

Syntax

```
boolean somFindMethod  (SOMClass  receiver, somId  methodId,
                        somMethodPtr  m)
```

Parameters

receiver (SOMClass)
A pointer to the class object whose method is desired.

methodId (somId)
An ID that represents the name of the desired method. The **somIdFromString** function can used to obtain an ID from the method's name.

m (somMethodPtr)
A pointer to the location in memory where a pointer to the specified method's procedure should be stored**.** The method stores a NULL pointer in this location (if the method does not exist) or a value that can be called.

Returns

rc (boolean)

TRUE	Returns TRUE when the method procedure can be called directly and FALSE when the method procedure is a dispatch function.
FALSE	Returns FALSE when the method procedure is a dispatch function.

Remarks

This method performs name-lookup method resolution, determines the method procedure appropriate for performing the indicated method on instances of the receiving class, and loads *m* with the method procedure address. For static methods, method procedure resolution is done using the instance method table of the receiving class.

Name-lookup resolution must be used to invoke dynamic methods. Also, name-lookup can be useful when different classes introduce methods of the same name, signature, and desired semantics, but it is not known until runtime which of these classes should be used as a type for the objects on which the method is to be invoked. If the signature of a method is a not known, then method procedures cannot be be used directly, and the **somDispatch** method can be used after dynamically discovering the signature to allow the correct arguments can be placed on a va_list.

As with any method that returns procedure pointers, this method allows repeated invocations of the same method procedure to be programmed. If this is done, it up to the programmer to prevent runtime errors by assuring that each invocation is performed either on an instance of the class used to resolve the method procedure or of some class derived from it. Whenever using SOM method procedure pointers, it is necessary to indicate the arguments to be passed and the use of system linkage to the compiler, so it can generate a correct procedure call. The way this is done depends on the compiler and the system being used. However, C and C++ usage bindings provide an appropriate typedef for static methods. The name of the typedef is based on the name of the class that introduces the method, as illustrated in the example below.

If the class does not support the specified method, then *m is set to NULL and the return value is meaningless. Otherwise, the returned result is true if the indicated method was a static method.

Original Class

SOMClass

Related Methods

Methods

- **somFindSMethod**
- **somFindSMethodOK**
- **somSupportsMethod**
- **somDispatch,**
- **somClassDispatch**

Functions

- **somApply**
- **somResolve**
- **somClassResolve**
- **somResolveByName**
- **somParentNumResolve**

Macros

- **SOM_Resolve**
- **SOM_ResolveNoCheck**
- **SOM_ParentNumResolve**

Example Code

Assuming that the *Animal* class introduces the method *setSound*, the type name for the *setSound* method procedure type will be *somTD_Animal_setSound*, as illustrated in the **Example**.

```
#include <animal.h>
void main()
{
   Animal myAnimal;
   somId somId_setSound;
   somTD_Animal_setSound methodPtr;
   Environment *ev = somGetGlobalEnvironment();

   myAnimal = AnimalNew();
/* ---------------------------------
   Note: Next three lines are equivalent to
       _setSound(myAnimal, ev, "Roar!!!");
   --------------------------------- */
   somId_setSound = somIdFromString("setSound");
   _somFindMethod (_somGetClass(myAnimal),
                   somId_setSound, &methodPtr);
   methodPtr(myAnimal, ev, "Roar!!!");
/* --------------------------------- */
   _display(myAnimal, ev);
   _somFree(myAnimal);
}
/*
Program Output:
This Animal says
Roar!!!
*/
```

somFindMethodOk

This method finds the method procedure for a method and indicates whether it represents a static method or a dynamic method. Not generally overridden.

For backward compatibility, this method does not take an **Environment** parameter.

Syntax

```
boolean somFindMethodOk  (SOMClass receiver, somId methodId,
                          somMethodPtr m)
```

Parameters

receiver (SOMClass)
A pointer to the class object whose method is desired.

methodId (somId)
An ID that represents the name of the desired method. The **somIdFromString** function can used to obtain an ID from the method's name.

m (somMethodPtr)
A pointer to the location in memory where a pointer to the specified method's procedure should be stored**.** Both methods store a NULL pointer in this location (if the method does not exist) or a value that can be called.

Returns

rc (boolean)

TRUE	Returns TRUE when the method procedure can be called directly and FALSE when the method procedure is a dispatch function.
FALSE	Returns FALSE when the method procedure is a dispatch function.

Remarks

This method performs name-lookup method resolution, determines the method procedure appropriate for performing the indicated method on instances of the receiving class, and loads *m* with the method procedure address. For static methods, method procedure resolution is done using the instance method table of the receiving class.

Name-lookup resolution must be used to invoke dynamic methods. Also, name-lookup can be useful when different classes introduce methods of the same name, signature, and desired semantics, but it is not known until runtime which of these classes should be used as a type for the objects on which the method is to be invoked. If the signature of a method is a not known, then method procedures cannot be be used directly, and the **somDispatch** method can be used after dynamically discovering the signature to allow the correct arguments can be placed on a va_list.

As with any method that returns procedure pointers, this method allows repeated invocations of the same method procedure to be programmed. If this is done, it up to the programmer to prevent runtime errors by assuring that each invocation is performed either on an instance of the class used to resolve the method procedure or of some class derived from it. Whenever using SOM method procedure pointers, it is necessary to indicate the arguments to be passed and the use of system linkage to the compiler, so it can generate a correct procedure call. The way this is done depends on the compiler and the system being used. However, C and C++ usage bindings provide an appropriate typedef for static methods. The name of the typedef is based on the name of the class that introduces the method, as illustrated in the example below.

Unlike the **somFindMethod** method, if the class does not support the specified method, the **somFindMethodOk** method raises an error and halts execution.

If the class does not support the specified method, then **m* is set to NULL and the return value is meaningless. Otherwise, the returned result is true if the indicated method was a static method.

Original Class

SOMClass

Related Methods

Methods

- **somFindSMethod**
- **somFindSMethodOK**
- **somSupportsMethod**
- **somDispatch,**
- **somClassDispatch**

Functions

- **somApply**
- **somResolve**
- **somClassResolve**
- **somResolveByName**

- **somParentNumResolve**

Macros

- **SOM_Resolve**
- **SOM_ResolveNoCheck**
- **SOM_ParentNumResolve**

Example Code

Assuming that the *Animal* class introduces the method *setSound*, the type name for the *setSound* method procedure type will be *somTD_Animal_setSound*, as illustrated in the **Example**.

```
#include <animal.h>
void main()
{
   Animal myAnimal;
   somId somId_setSound;
   somTD_Animal_setSound methodPtr;
   Environment *ev = somGetGlobalEnvironment();

   myAnimal = AnimalNew();
/* ---------------------------------
   Note: Next three lines are equivalent to
       _setSound(myAnimal, ev, "Roar!!!");
   --------------------------------- */
   somId_setSound = somIdFromString("setSound");
   _somFindMethod (_somGetClass(myAnimal),
                   somId_setSound, &methodPtr);
   methodPtr(myAnimal, ev, "Roar!!!");
/* --------------------------------- */
   _display(myAnimal, ev);
   _somFree(myAnimal);
}
/*
Program Output:
This Animal says
Roar!!!
*/
```

somFindSMethod

This method finds the method procedure for a static method. Not generally overridden.

For backward compatibility, this method does *not* take an **Environment** parameter.

Syntax

```
somMethodPtr somFindSMethod   (SOMClass  receiver, somId  methodId)
```

Parameters

receiver (SOMClass)
A pointer to a class object.

methodId (somId)
A somId representing the name of the desired method.

Returns

rc (somMethodPtr)

This method returns a pointer to the method procedure that supports the specified method for the class.

Remarks

This method performs name-lookup resolution in a similar fashion to **somFindMethod** and **somFindMethodOK**, but are restricted to static but is restricted to static methods. See the description of **somFindMethod** for a discussion of name-lookup method resolution. Because this method is restricted to resolving static methods, its interface is slightly different from the **somFindMethod** interface; a method procedure pointer is returned when lookup is successful; otherwise NULL is returned.

The **somFindSMethodOk** method is identical to this method except that, with **somFindSMethodOk**, an error is raised if the indicated static method is not defined for the receiving class, and execution is halted.

Original Class

SOMClass

Related Methods

Methods

- **somFindMethod**
- **somFindMethodOk**

Example Code

```
See the somFindMethod method example.
```

somFindSMethodOk

This method finds the method procedure for a static method. Not generally overridden.

For backward compatibility, this method does *not* take an **Environment** parameter.

Syntax

```
somMethodPtr somFindSMethodOk  (SOMClass  receiver,
                                somId  methodId)
```

Parameters

receiver (SOMClass)
A pointer to a class object.

methodId (somId)
A somId representing the name of the desired method.

Returns

rc (somMethodPtr)

This method returns a pointer to the method procedure that supports the specified method for the class.

Remarks

This method performs name-lookup resolution in a similar fashion to **somFindMethod** and **somFindMethodOK**, but are restricted to static but is restricted to static methods. See the description of **somFindMethod** for a discussion of name-lookup method resolution. Because this method is restricted to resolving static methods, its interface is slightly different from the **somFindMethod** interface; a method procedure pointer is returned when lookup is successful; otherwise NULL is returned.

This method is identical to **somFindSMethod**, except that an error is raised if the indicated static method is not defined for the receiving class, and execution is halted.

Original Class

SOMClass

Related Methods

Methods

- **somFindMethod**
- **somFindMethodOk**

Example Code

```
See the somFindMethod method example.
```

somGetInstancePartSize

This method returns the total size of the instance data structure introduced by a class. Not generally overridden.

For backward compatibility, this method does *not* take an **Environment** parameter.

Syntax

```
long somGetInstancePartSize   (SOMClass  receiver)
```

Parameters

receiver (SOMClass)
A pointer to the class object whose instance data size is desired.

Returns

rc (long)

bytes	Returns the size, in bytes, of the instance variables introduced by this class. This does not include the size of instance variables introduced by this class's ancestor or descendent classes.
0	Returns 0 if a class introduces no instance variables.

Remarks

This method returns the amount of space needed in an object of the specified class or any of its subclasses to contain the instance variables introduced by the class.

Original Class

SOMClass

Related Methods

Methods

- **somGetInstanceSize**

Example Code

```
#include <animal.h>
main()
{
  Animal myAnimal;
  SOMClass animalClass;
  int instanceSize;
  int instanceOffset;
  int instancePartSize;

  myAnimal = AnimalNew ();
  animalClass = _somGetClass (myAnimal);
  instanceSize = _somGetInstanceSize (animalClass);
  instanceOffset = _somGetInstanceOffset (animalClass);
  instancePartSize = _somGetInstancePartSize (animalClass);
  somPrintf ("Instance Size: %d\n", instanceSize);
  somPrintf ("Instance Offset: %d\n", instanceOffset);
  somPrintf ("Instance Part Size: %d\n", instancePartSize);
  _somFree (myAnimal);
}
/*
Output from this program:
Instance Size: 8
Instance Offset: 0
Instance Part Size: 4
*/
```

somGetInstanceSize

This method returns the size of an instance of a class. Not generally overridden.

For backward compatibility, this method does *not* take an **Environment** parameter.

Syntax

```
long somGetInstanceSize   (SOMClass  receiver)
```

Parameters

receiver (SOMClass)
A pointer to the class object whose instance size is desired.

Returns

rc (long)

bytes Returns the size, in bytes, of each instance of this class. This includes the space required for instance variables introduced by this class and all of its ancestor classes.

Remarks

This method returns the total amount of space needed in an instance of the specified class.

Original Class

SOMClass

Related Methods

Methods

- **somGetInstancePartSize**

Example Code

```
#include <animal.h>
main()
{
  Animal myAnimal;
  SOMClass animalClass;
  int instanceSize;
  int instanceOffset;
  int instancePartSize;

       myAnimal = AnimalNew ();
  animalClass = _somGetClass (myAnimal);
  instanceSize = _somGetInstanceSize (animalClass);
  instanceOffset = _somGetInstanceOffset (animalClass);
  instancePartSize = _somGetInstancePartSize (animalClass);
  somPrintf ("Instance Size: %d\n", instanceSize);
  somPrintf ("Instance Offset: %d\n", instanceOffset);
  somPrintf ("Instance Part Size: %d\n", instancePartSize);
  _somFree (myAnimal);
}
/*
Output from this program:
Instance Size: 8
Instance Offset: 0
Instance Part Size: 4
*/
```

somGetInstanceToken

This method returns a data access token for the instance data introduced by a class.

For backward compatibility, this method does *not* take an **Environment** parameter.

Syntax

```
somDToken somGetInstanceToken  (SOMClass  receiver)
```

Parameters

receiver (SOMClass)
A pointer to a **SOMClass** object.

Returns

rc (somDToken)

Returns a data token for the beginning of the instance data introduced by the receiver.

Remarks

This method returns a data token "pointing" to the beginning of the instance data introduced by the receiving class. This token can be passed to the function **somDataResolve** to locate this instance data within an an instance of the receiver class or any class derived from it. Also the instance data token for a class can be passed to the class method **somGetMemberToken** to get a data token for a specific instance variables introduced by the class (if the relative offset of this instance variable is known). This approach is used by C and C++ implementation bindings to support public instance data for OIDL classes (IDL classes currently have no public instance data).

A data token for the instance data introduced by a class is required by method procedures that access data introduced by the method procedure's defining class. For classes declared using OIDL and IDL, the needed token is stored in the auxiliary class data structure, which is an external data structure made statically available by the C and C++ language bindings as <className>CClassData.instanceToken. Thus, this method call is not generally used by C and C++ class implementors of classes declared using OIDL or IDL.

Original Class

SOMClass

Related Methods

Functions

- **somDataResolve**

Methods

- **somGetInstanceSize**
- **somGetInstancePartSize**
- **somGetMemberToken**

somGetMemberToken

This method returns an access token for an instance variable. This method is not generally overridden.

For backward compatibility, this method does *not* take an **Environment** parameter.

Syntax

```
somDToken somGetMemberToken  (SOMClass receiver,
                              long memberOffset,
                              somDToken instanceToken)
```

Parameters

receiver (SOMClass)
A pointer to a **SOMClass** object.

memberOffset (long)
A 32-bit integer representing the offset of the required data member.

instanceToken (somDToken)
A token, obtained from **somGetInstanceToken**, that identifies the introduced portion of the class.

Returns

rc (somDToken)

Returns an access token for the specified data member.

Remarks

This method returns an access token for the data member at offset *memberOffset* within the block of instance data identified by instance token. The returned token can subsequently be passed to the **somDataResolve** function to locate the data member.

Typically, only the code that implements a class declared using OIDL requires access to this method, and this code is normally provided by implementation bindings. Thus C and C++ programmers do not normally invoke this method.

Original Class

SOMClass

Related Methods

Functions

- **somDataResolve**

Methods

- **somGetInstanceSize**
- **somGetInstancePartSize**
- **somGetInstanceToken**

somGetMethodData

This method returns method information for a specified method, which must have been introduced by the receiver class or an ancestor of that class. Not generally overridden.

For backward compatibility, this method does *not* take an **Environment** parameter.

Syntax

```
boolean somGetMethodData  (SOMClass  receiver, somId  methodId,
                           somMethodData  md)
```

Parameters

receiver (SOMClass)
A pointer to the class that produced the index value.

methodId (somId)
A **somId** for the method's name.

md (somMethodData)
A pointer to a *somMethodData* structure.

Returns

rc (boolean)

true	Returns boolean true if successful.
false	Returns false if not successful.

Remarks

This method loads a *somMethodData* structure with data describing the method identified by the passed *methodId*. If *methodId* does not identify a method known to the receiver, then false is returned; otherwise, true is returned after loading the *somMethodData* structure with data corresponding to the indicated method.

Original Class

SOMClass

Related Methods

Data Structures

- **somMethodData (somapi.h)**

Methods

- **somGetMethodIndex**
- **somGetMethodData**
- **somGetNthMethodInfo**

Example Code

```
#include <somcls.xh>

main
{
   somEnvironmentNew();
   somId gmiId = somIdFromString("somGetMethodIndex");
   somMethodData md;
   boolean rc = _SOMClass->somGetMethodData(gmiId, &md);
   SOM_Test(rc && somCompareIds(gmiId, md.id);
}
```

somGetMethodDescriptor

This method returns the method descriptor for a method. Not generally overridden.

For backward compatibility, this method does *not* take an **Environment** parameter.

Syntax

```
somId somGetMethodDescriptor  (SOMClass  receiver, somId  methodId)
```

Parameters

receiver (SOMClass)
A pointer to a **SOMClass**.

methodId (somId)
A **somId** method descriptor.

Returns

rc (somId)

Returns a **somId** method descriptor.

Remarks

This method returns the method descriptor for a specified method of a class. (A method descriptor is a somId that represents the identifier of an attribute definition or a method definition in the SOM Interface Repository. It contains information about the method's return type and the types of its arguments.) If the class object does not support the indicated method, NULL is returned.

Original Class

SOMClass

Related Methods

Methods

- **somAddDynamicMethod**
- **somGetNthMethodInfo**
- **somGetMethodData**
- **somGetNthMethodData**

Example Code

```
somId myMethodDescriptor;
myMethodDescriptor = _somGetMethodDescriptor(_Animal,
                                 somIdFromString("setSound"));
```

somGetMethodIndex

This method returns a class-specific index for a method. Not generally overridden.

For backward compatibility, this method does *not* take an **Environment** parameter.

Syntax

```
long somGetMethodIndex  (SOMClass  receiver, somId  methodId)
```

Parameters

receiver (SOMClass)
A pointer to a **SOMClass** object.

methodId (somId)
A somId method ID.

Returns

rc (long)

long	Returns a positive long if successful.
-1	Returns a -1 if not successful.

Remarks

This method returns an index that can be used in subsequent calls to the same receiving class to determine information about the indicated (static or dynamic) method, via the methods **somGetNthMethodData** and **somGetNthMethodInfo**. The method must be appropriate for use on an instance of the receiver class; otherwise, a -1 is returned. The index of a method can change over time if dynamic methods are added to the receiver class or its ancestors. Thus, in dynamic multi-threaded environments, a critical region should be used to bracket the use of this method and of subsequent requests for method information based on the returned index.

Original Class

SOMClass

Related Methods

Data Structures

- **somMethodData (somapi.h)**

Methods

- **somGetNthMethodData**
- **somGetNthMethodInfo**

Example Code

```
#include <somcls.xh>
main
{
   somEnvironmentNew();
   somId gmiId = somIdFromString("somGetMethodIndex");
   long index = _SOMClass->somGetMethodIndex(gmiId);
   somMethodData md;
   boolean rc = _SOMClass->somGetNthMethodData(index,&md);
   SOM_Test(rc && somCompareIds(gmiId, md.id));
}
```

somGetMethodToken

This method returns a method access token for static method. Not generally overridden.

For backward compatibility, this method does *not* take an **Environment** parameter.

Syntax

```
somMToken somGetMethodToken  (SOMClass  receiver,
                              somId  methodId)
```

Parameters

receiver (SOMClass)
A pointer to a **SOMClass** object.

methodId (somId)
A **somId** identifying a method.

Returns

rc (somMToken)

Returns a **somMToken** method access token.

Remarks

This method returns a method access token for a static method with the specified id that was introduced by the receiver class or an ancestor of the receiver class. This method token can be passed to the **somResolve** function (or one of the other offset-based method resolution functions) to select a method procedure pointer from a method table of an object whose class is the same as, or is derived from the class that introduced the method.

The example below assumes that the class *Animal* introduces the method *setSound*.

Original Class

SOMClass

Related Methods

Functions

- **somResolve**
- **somClassResolve**
- **somParentNumResolve**

Methods

- **somGetNthMethodInfo**
- **somGetMethodData**

Example Code

```
#include < >
main() {
somMToken tok;
Animal myAnimal;
somTD_Animal_setSound methodPtr; /* use typedef from animal.h */
Environment *ev = somGetGlobalEnvironment();
myAnimal = AnimalNew();

/* next 3 lines equivalent to _setSound(myAnimal, ev, "Roar!!!"); */
tok = _ somGetMethodToken(_Animal, somIdFromString("setSound"));
methodPtr = (somTD_Animal_setSound)somResolve(myAnimal, tok);
methodPtr(myAnimal, ev, "Roar!!!");
_display(myAnimal, ev);
_somFree(myAnimal);
}
```

somGetname

This method returns the name of a class. Not generally overridden.

For backward compatibility, this method does *not* take an **Environment** parameter.

Syntax

```
string somGetname   (SOMClass  receiver)
```

Parameters

receiver (SOMClass)
The class whose name is desired.

Returns

rc (string)
Returns a pointer to the name of the class.

Remarks

This method returns the address of a zero-terminated string that gives the name of the receiving class. This name may be used as a RepositoryId in the **Repository_lookup_id** method (described in the SOM Interface Repository Framework section) to obtain the IDL interface definition that corresponds to the receiving class.

The returned name is not necessarily the same as the statically known class name used by a programmer to gain access to the class object (e.g., via the method **somFindClass**). This is because the method **somSubstituteClass** may have been used to "shadow" the class having the static name used by the programmer.

Also, when the interface to a class's instances is defined within an IDL module, the returned name will not directly correspond to the names of the procedures and macros made available by C and C++ usage bindings for accessing class objects (e.g., the **<className>NewClass** procedure, or the **_<className>** macro). This is because the token used in constructing the names of these procedures and macros uses an underscore character to separate the module name from the interface name, while the actual name of the corresponding class uses two colon characters instead of an underscore for this purpose.

This method is not generally overridden. The returned address is valid until the class object is unregistered or freed.

Original Class

SOMClass

Related Methods

Methods

- **Repository_lookup_id**
- **somSubstituteClass**
- **somFindClass**

Example Code

```
#include <animal.h>   /* assume Animal defined in th        e Zoo module */
#include <string.h>
main()
{
  string className = Zoo_AnimalNewClass(0,0)->somGetName();
  SOM_Test(!strcmp(  className, "Zoo::Animal"));
}
```

somGetNthMethodData

This method returns method information for the *n*th (static or dynamic) method known to a given class. Not generally overridden.

For backward compatibility, this method does *not* take an **Environment** parameter.

Syntax

```
boolean somGetNthMethodData  (SOMClass  receiver, long  index,
                              somMethodData  md)
```

Parameters

receiver (SOMClass)
A pointer to the class that produced the index value.

index (long)
An index returned as a result of a previous call of **somGetMethodIndex**.

md (somMethodData)
A pointer to a *somMethodData* structure.

Returns

rc (boolean)

true	Returns boolean true if successful.
false	Returns false if not successful.

Remarks

This method loads a *somMethodData* structure with data describing the method identified by the passed index. The index must have been produced by a previous call to exactly the same receiver class; the same method will in general have different indexes in different classes. If the index does not identify a method known to this class, then false is returned; otherwise, true is returned after loading the *somMethodData* structure with data corresponding to the indicated method.

Related Methods

Data Structures

somMethodData (somapi.h)

Methods

- **somGetMethodIndex**
- **somGetMethodData**
- **somGetNthMethodInfo**

Example Code

```
#include <somcls.xh>
main
{
   somEnvironmentNew();
   somId gmiId = somIdFromString("somGetMethodIndex");
   long index = _SOMClass->somGetMethodIndex(gmiId);
   somMethodData md;
   boolean rc = _SOMClass->somGetNthMethodData(index,&md);
   SOM_Test(rc && somCompareIds(gmiId, md.id));
}
```

somGetNthMethodInfo

This method returns the **somId** of the *n*th (static or dynamic) method known to a given class. Also loads a **somId** with a descriptor for the method. Not generally overridden.

For backward compatibility, this method does *not* take an **Environment** parameter.

Syntax

```
somId somGetNthMethodInfo  (SOMClass  receiver, long  index,
                            somId  descriptor)
```

Parameters

receiver (SOMClass)
A pointer to the class from which the *index* was obtained using method **somGetMethodIndex**.

index (long)
The *n*th method known to this class, whose method descriptor is desired.

descriptor (somId)
A pointer to a **somId** that will be loaded with a **somId** for the descriptor.

Returns

rc (somId)

somID	Returns the **somId** for the indicated method, if a method with the indicated index is known to the receiver.
NULL	Returns NULL if a method with the indicated index is not known to the receiver.

Remarks

This method returns the identifier of a method, and loads the **somId** whose address is passed with the **somId** of the method descriptor. Method descriptors are used to support access to information stored in a SOM Interface Repository.

Original Class

SOMClass

Related Methods

Classes

- **Repository (repostry.idl)**

Methods

- **somGetMethodIndex**
- **somGetNthMethodData**

Example Code

```
#include <somcls.xh>
main()
{
  somEnvironmentNew();
  somId descriptor, icId = somIdFromSring("somGetMethodIndex");
  long ndx = _SOMClass->somGetMethodIndex(icId);
  somMethodData md;
  boolean rc = _SOMClass->somGetNthMethodData(index,&md);
  SOM_Test(rc && somCompareIds(gmiId, md.id));
}
```

somGetNumMethods

This method returns the number of methods available for a class of objects. Not generally overridden.

For backward compatibility, this method does *not* take an **Environment** parameter.

Syntax

```
long somGetNumMethods   (SOMClass  receiver)
```

Parameters

receiver (SOMClass)
A pointer to the class object whose instance method count is desired.

Returns

rc (long)
Returns the total number of methods that are currently available for the receiving class.

Remarks

This method returns the number of methods currently supported by the specified class, including inherited methods (both static and dynamic).

The value that this method returns is the total number of methods currently known to the receiving class as being applicable to its instances. This includes both static and dynamic methods, whether defined in this class or inherited from an ancestor class.

Original Class

SOMClass

Related Methods

Methods

- **somGetNumStaticMethods**

Example Code

```
#include <animal.h>
main()
{
    long numMethods;

    AnimalNewClass(Animal_MajorVersion, Animal_MinorVersion);
    numMethods = _somGetNumMethods(_Animal);
    somPrintf("Number of methods supported by class: %d\n",
                                                    numMethods);
}
```

somGetNumStaticMethods

This method obtains the number of static methods available for a class of objects. Not generally overridden.

For backward compatibility, this method does *not* take an **Environment** parameter.

Syntax

```
long somGetNumStaticMethods   (SOMClass  receiver)
```

Parameters

receiver (SOMClass)
A pointer to the class object whose static method count is desired.

Returns

rc (long)
Returns the total number of static methods that are available for instances of the receiving class.

Remarks

This method returns the number of static methods available in the specified class, including inherited ones. Static methods are those that are represented by entries in the class's instance method table, and which can be invoked using method tokens and offset resolution.

Original Class

SOMClass

Related Methods

Methods

- **somGetNumMethods**

Example Code

```
#include <animal.h>
main()
{
    long numMethods;

    AnimalNewClass(Animal_MajorVersion, Animal_MinorVersion);
    numMethods = _somGetNumStaticMethods(_Animal);
    somPrintf("Number of static methods supported by class: %d\n",
                numMethods);
}
```

somGetParents

This method gets a pointer to a class's parent (direct base) classes. Not generally overridden. Note: For backward compatibility, this method does *not* take an Environment parameter.

Syntax

```
SOMClassSequence somGetParents   (SOMObject  receiver)
```

Parameters

receiver (SOMObject)
A pointer to the class whose parent (base) classes are desired.

Returns

rc (SOMClassSequence)
The **somGetParents** method returns a sequence of pointers to the parents of the receiver, or NULL otherwise (in the case of **SOMObject**). The sequence of parents is in left-to-right order.

Remarks

The **somGetparents** method returns a sequence containing pointers to the parents of the receiver.

Original Class

SOMClass

Related Methods

Methods

- **somGetClass**

Example Code

```
/* Note: Dog is a single-inheritance subclass of Animal. */
#include <dog.idl>
main ()
{
   Dog myDog;
   SOMClass dogClass;
   SOMClassSequence parents;
   char *parentName;
   int i;

   myDog = DogNew();
   dogClass = _somGetClass(myDog);
   parents = _somGetPaarents(dogClass);
   for (i=0; i<parents._length; i++)
      somprintf("-- parent %d is %s\n", i;
                _somGetName(parents._buffer[i]));
   _somFree(myDog);
}
/*
Output from this program:
-- parent 0 is Animal
*/
```

somGetVersionNumbers

This method gets the major and minor version numbers of a class. Not generally overridden.

For backward compatibility, this method does *not* take an **Environment** parameter.

Syntax

```
void somGetVersionNumbers (SOMClass receiver, long majorVersion,
                           long minorVersion)
```

Parameters

receiver (SOMClass)
A pointer to a class object.

majorVersion (long)
A pointer where the major version number is to be stored.

minorVersion (long)
A pointer where the minor version number is to be stored.

Returns

rc (void)

Remarks

This method returns, via its output parameters, the major and minor version numbers of the class specified by *receiver*. The class object must have already been created (because the class object is the receiver of the method).

Original Class

SOMClass

Related Methods

Methods

- **somCheckVersion**

Example Code

```
#include <som.h>

main() {

  long major, minor;
  SOMClass myClass;

  somEnvironmentNew();
  myClass = _somFindClass(SOMClassMgrObject,
                      somIdFromString("Animal"), 0, 0);
  _somGetVersionNumbers(myClass, &major, &minor);
  somPrintf("The version numbers are %i and %i.\n", major, minor);
}
```

somLookupMethod

Performs name-look method resolution. Not generally overridden.

For backward compatibility, this method does *not* take an **Environment** parameter.

Syntax

```
somMethodPtr somLookupMethod  (SOMClass  receiver,
                               somId  methodId)
```

Parameters

receiver (SOMClass)
A pointer to the class whose instance method for the indicated method is desired.

methodId (somId)
A **somId** of the method whose method-procedure pointer is needed.

Returns

rc (somMethodPtr)
A pointer to the method procedure that supports the method indicated by *methodId*. Or, if the method is not supported by the receiving class, then an error is returned, and execution is halted.

Remarks

The **somLookupMethod** method uses name-lookup resolution to return the address of the method procedure that supports the indicated method on instances of the receiver class. The method may be either static or dynamic. The SOM C and C++ usage bindings support name-lookup method resolution by invoking **somLookupMethod** on the class of the object on which a name-lookup method invocation is made.

The **somLookupMethod** method is like **somFindSMethodOK** except that dynamic methods can also be returned. If the method is not supported by the receiving class, then an error is returned and execution is halted. To check the existence of a method, **somFindMethod** can be used.

As always, in order to use a method procedure pointer such as that returned by **somLookupMethod**, it is necessary to typecast the procedure pointer so that the compiler can create the correct procedure call. This means that a programmer making

explicit use of this method must either know the signature of the identified method, and from this create a typedef indicating system linkage and the appropriate argument and return types, or make use of an existing typedef provided by C or C++ usage bindings for a SOM class that introduces a static method with the desired signature.

Original Class

SOMClass

Related Methods

Methods

- **somFindSMethod**
- **somFindSMethodOk**
- **somFindMethod**
- **somFindMethodOk**

Example Code

```
#include <somcls.xh>
#include <somcm.xh>
void main()
{
   somId fcpId = somIdFromString("somFindClass")
   somId animalId = somIdFromString("Animal");
   SOMClassMgr *cm = somEnvironmentNew();
   somTD_SOMClassMgr_somFindClass findclassproc =
         (somTD_SOMClassMgr_somFindClass)
             _SOMClassMgr->somLookupMethod(fcpId);
   SOMClass *aCls = findclassproc(cm,animalId,0,0);
   ...
   somFree(fcpId);
   somFree(animalId);
}
```

somNew

This method creates a new instance of a class.

For backward compatibility, this method does *not* take an **Environment** parameter.

Syntax

```
SOMObject somNew   (SOMClass  receiver)
```

Parameters

receiver (SOMClass)
A pointer to the class object that is to create a new instance.

Returns

rc (SOMObject)
A pointer to the newly created SOMObject object, or NULL.

Remarks

The **somNew** and **somNewNoInit** methods create a new instance of the receiving class. Space is allocated as necessary to hold the new object.

When either of these methods is applied to a class, the result is a new instance of that class. If the receiver class is **SOMClass** or a class derived from SOMClass, the new object will be a class object; otherwise, the new object will not be a class object. The **somNew** method invokes the **somDefaultInit** method on the newly created object. The **somNewNoInit** method does not.

Either method can fail to allocate enough memory to hold a new object and, if so, NULL is returned.

The SOM Compiler generates convenience macros for creating instances of each class, for use by C and C++ programmers. These macros can be used in place of this method.

Original Class

SOMClass

Related Methods

Methods

- **somRenew**

Example Code

```
#include <animal.h>

void main()
{    Animal myAnimal;
/*-----------------------------------------
Note:  next 2 lines are functionally equivalent to
         myAnimal = AnimalNew();
----------------------------------------- */
    /* Create class object: */
    AnimalNewClass(Animal_MajorVersion, AnimalMinorVersion);
    myAnimal = _somNew(_Animal);  /* Create instance of Animal cls */
    /* ... */
    _somFree(myAnimal);       /* Free instance of Animal */
 }
```

somNewNoInit

This method creates a new instance of a class.

For backward compatibility, this method does *not* take an **Environment** parameter.

Syntax

```
SOMObject somNewNoInit  (SOMClass  receiver)
```

Parameters

receiver (SOMClass)
A pointer to the class object that is to create a new instance.

Returns

rc (SOMObject)
A pointer to the newly created SOMObject object, or NULL.

Remarks

The **somNew** and **somNewNoInit** methods create a new instance of the receiving class. Space is allocated as necessary to hold the new object.

When either of these methods is applied to a class, the result is a new instance of that class. If the receiver class is **SOMClass** or a class derived from SOMClass, the new object will be a class object; otherwise, the new object will not be a class object. The **somNew** method invokes the **somDefaultInit** method on the newly created object. The **somNewNoInit** method does not.

Either method can fail to allocate enough memory to hold a new object and, if so, NULL is returned.

The SOM Compiler generates convenience macros for creating instances of each class, for use by C and C++ programmers. These macros can be used in place of this method.

Original Class

SOMClass

Related Methods

Methods

- **somRenew**

Example Code

```
#include <animal.h>

void main()
{    Animal myAnimal;
/*-----------------------------------------
Note:  next 2 lines are functionally equivalent to
         myAnimal = AnimalNew();
----------------------------------------- */
    /* Create class object: */
    AnimalNewClass(Animal_MajorVersion, AnimalMinorVersion);
    myAnimal = _somNew(_Animal);  /* Create instance of Animal cls */
    /* ... */
    _somFree(myAnimal);       /* Free instance of Animal */
 }
```

somRenew

This method creates a new object instance using a passed block of storage.

For backward compatibility, this method does *not* take an **Environment** parameter.

Syntax

```
SOMObject somRenew   (SOMClass  receiver, somToken  memPtr)
```

Parameters

receiver (SOMClass)
A pointer to the class object that is to create the new instance.

memPtr (somToken)
A pointer to the space to be used to construct a new object.

Returns

rc (SOMObject)
The value of *newObject* is returned, which is now a pointer to a valid, initialized object.

Remarks

The **somRenew** method creates a new instance of the receiving class by setting the appropriate location in the passed memory block to the receiving class's instance method table. Unlike **somNew**, these "Renew" methods use the space pointed to by *memPtr* rather than allocating new space for the object. The **somRenew** method automatically re-initializes the object by first zeroing the object's memory, and then invoking **somDefaultInit**; **somRenewNoInit** zeros memory, but does not invoke **somDefaultInit**. **somRenewNoInitNoZero** only sets the method table pointer; while **somRenewNoZero** calls **somDefaultInit**, but does not zero memory first.

No check is made to ensure that the passed pointer addresses enough space to hold an instance of the receiving class. The caller can determine the amount of space necessary by using the **somGetInstanceSize** method.

The C bindings produced by the SOM Compiler contain a macro that is a convenient shorthand for **_somRenew(***_className***)**.

Original Class

SOMClass

Related Methods

Methods

- **somGetInstanceSize**
- **somDefaultInit**
- **somNew**

Example Code

```
#include <animal.h>

main()
{
  void *myAnimalCluster;
  Animal animals[5];
  SOMClass animalClass;
  int animalSize, i;

  animalClass =
      AnimalNewClass(Animal_MajorVersion,Animal_MinorVersion);
  animalSize = _somGetInstanceSize (animalClass);
  /* Round up to double-word multiple */
  animalSize = ((animalSize+3)/4)*4;
  /*
   * Next line allocates room for 5 objects
   * in a "cluster" with a single memory-
   * allocation operation.
   */
  myAnimalCluster = SOMMalloc (5*animalSize);
  /*
   * The for-loop that follows creates 5 initialized
   * Animal instances within the memory cluster.
   */
  for (i=0; i
     animals[i] =
     _somRenew(animalClass, myAnimalCluster+(i*animalSize));
  /* Uninitialize the animals explicitly: */
  for (i=0; i
     _somUninit(animals[i]);
  /*
   * Finally, the next line frees all 5 animals
   * with one operation.
   */
  SOMFree (myAnimalCluster);
}
```

somRenewNoInit

This method creates a new object instance using a passed block of storage.

For backward compatibility, this method does *not* take an **Environment** parameter.

Syntax

```
SOMObject somRenewNoInit  (SOMClass  receiver, somToken  memPtr)
```

Parameters

receiver (SOMClass)
A pointer to the class object that is to create the new instance.

memPtr (somToken)
A pointer to the space to be used to construct a new object.

Returns

rc (SOMObject)
The value of *newObject* is returned, which is now a pointer to a valid, initialized object.

Remarks

The **somRenew** method creates a new instance of the receiving class by setting the appropriate location in the passed memory block to the receiving class's instance method table. Unlike **somNew**, these "Renew" methods use the space pointed to by *memPtr* rather than allocating new space for the object. The **somRenew** method automatically re-initializes the object by first zeroing the object's memory, and then invoking **somDefaultInit**; **somRenewNoInit** zeros memory, but does not invoke **somDefaultInit**. **somRenewNoInitNoZero** only sets the method table pointer; while **somRenewNoZero** calls **somDefaultInit**, but does not zero memory first.

No check is made to ensure that the passed pointer addresses enough space to hold an instance of the receiving class. The caller can determine the amount of space necessary by using the **somGetInstanceSize** method.

The C bindings produced by the SOM Compiler contain a macro that is a convenient shorthand for **_somRenew(***_className***)**.

somRenewNoInit

Original Class

SOMClass

Related Methods

Methods

- **somGetInstanceSize**
- **somDefaultInit**
- **somNew**

Example Code

```
#include <animal.h>

main()
{
  void *myAnimalCluster;
  Animal animals[5];
  SOMClass animalClass;
  int animalSize, i;

  animalClass =
      AnimalNewClass(Animal_MajorVersion,Animal_MinorVersion);
  animalSize = _somGetInstanceSize (animalClass);
  /* Round up to double-word multiple */
  animalSize = ((animalSize+3)/4)*4;
  /*
   * Next line allocates room for 5 objects
   * in a "cluster" with a single memory-
   * allocation operation.
   */
  myAnimalCluster = SOMMalloc (5*animalSize);
  /*
   * The for-loop that follows creates 5 initialized
   * Animal instances within the memory cluster.
   */
  for (i=0; i
     animals[i] =
     _somRenew(animalClass, myAnimalCluster+(i*animalSize));
  /* Uninitialize the animals explicitly: */
  for (i=0; i
     _somUninit(animals[i]);
  /*
   * Finally, the next line frees all 5 animals
   * with one operation.
   */
  SOMFree (myAnimalCluster);
}
```

somRenewNoInitNoZero

This method creates a new object instance using a passed block of storage.

For backward compatibility, this method does *not* take an **Environment** parameter.

Syntax

```
SOMObject somRenewNoInitNoZero  (SOMClass  receiver,
                                 somToken  memPtr)
```

Parameters

receiver (SOMClass)
A pointer to the class object that is to create the new instance.

memPtr (somToken)
A pointer to the space to be used to construct a new object.

Returns

rc (SOMObject)
The value of *newObject* is returned, which is now a pointer to a valid, initialized object.

Remarks

The **somRenew** method creates a new instance of the receiving class by setting the appropriate location in the passed memory block to the receiving class's instance method table. Unlike **somNew**, these "Renew" methods use the space pointed to by *memPtr* rather than allocating new space for the object. The **somRenew** method automatically re-initializes the object by first zeroing the object's memory, and then invoking **somDefaultInit**; **somRenewNoInit** zeros memory, but does not invoke **somDefaultInit**. **somRenewNoInitNoZero** only sets the method table pointer; while **somRenewNoZero** calls **somDefaultInit**, but does not zero memory first.

No check is made to ensure that the passed pointer addresses enough space to hold an instance of the receiving class. The caller can determine the amount of space necessary by using the **somGetInstanceSize** method.

The C bindings produced by the SOM Compiler contain a macro that is a convenient shorthand for **_somRenew(***_className***)**.

somRenewNoInitNoZero

Original Class

SOMClass

Related Methods

Methods

- **somGetInstanceSize**
- **somDefaultInit**
- **somNew**

somRenewNoInitNoZero

Example Code

```
#include <animal.h>

main()
{
  void *myAnimalCluster;
  Animal animals[5];
  SOMClass animalClass;
  int animalSize, i;

  animalClass =
      AnimalNewClass(Animal_MajorVersion,Animal_MinorVersion);
  animalSize = _somGetInstanceSize (animalClass);
  /* Round up to double-word multiple */
  animalSize = ((animalSize+3)/4)*4;
  /*
   * Next line allocates room for 5 objects
   * in a "cluster" with a single memory-
   * allocation operation.
   */
  myAnimalCluster = SOMMalloc (5*animalSize);
  /*
   * The for-loop that follows creates 5 initialized
   * Animal instances within the memory cluster.
   */
  for (i=0; i
     animals[i] =
     _somRenew(animalClass, myAnimalCluster+(i*animalSize));
  /* Uninitialize the animals explicitly: */
  for (i=0; i
     _somUninit(animals[i]);
  /*
   * Finally, the next line frees all 5 animals
   * with one operation.
   */
  SOMFree (myAnimalCluster);
}
```

somRenewNoZero

This method creates a new object instance using a passed block of storage.

For backward compatibility, this method does *not* take an **Environment** parameter.

Syntax

```
SOMObject somRenewNoZero   (SOMClass  receiver, somToken  memPtr)
```

Parameters

receiver (SOMClass)
A pointer to the class object that is to create the new instance.

memPtr (somToken)
A pointer to the space to be used to construct a new object.

Returns

rc (SOMObject)
The value of *newObject* is returned, which is now a pointer to a valid, initialized object.

Remarks

The **somRenew** method creates a new instance of the receiving class by setting the appropriate location in the passed memory block to the receiving class's instance method table. Unlike **somNew**, these "Renew" methods use the space pointed to by *memPtr* rather than allocating new space for the object. The **somRenew** method automatically re-initializes the object by first zeroing the object's memory, and then invoking **somDefaultInit**; **somRenewNoInit** zeros memory, but does not invoke **somDefaultInit**. **somRenewNoInitNoZero** only sets the method table pointer; while **somRenewNoZero** calls **somDefaultInit**, but does not zero memory first.

No check is made to ensure that the passed pointer addresses enough space to hold an instance of the receiving class. The caller can determine the amount of space necessary by using the **somGetInstanceSize** method.

The C bindings produced by the SOM Compiler contain a macro that is a convenient shorthand for **_somRenew(***_className***)**.

somRenewNoZero

Original Class

SOMClass

Related Methods

Methods

- **somGetInstanceSize**
- **somDefaultInit**
- **somNew**

Example Code

```
#include <animal.h>

main()
{
  void *myAnimalCluster;
  Animal animals[5];
  SOMClass animalClass;
  int animalSize, i;

  animalClass =
      AnimalNewClass(Animal_MajorVersion,Animal_MinorVersion);
  animalSize = _somGetInstanceSize (animalClass);
  /* Round up to double-word multiple */
  animalSize = ((animalSize+3)/4)*4;
  /*
   * Next line allocates room for 5 objects
   * in a "cluster" with a single memory-
   * allocation operation.
   */
  myAnimalCluster = SOMMalloc (5*animalSize);
  /*
   * The for-loop that follows creates 5 initialized
   * Animal instances within the memory cluster.
   */
  for (i=0; i
     animals[i] =
     _somRenew(animalClass, myAnimalCluster+(i*animalSize));
  /* Uninitialize the animals explicitly: */
  for (i=0; i
     _somUninit(animals[i]);
  /*
   * Finally, the next line frees all 5 animals
   * with one operation.
   */
  SOMFree (myAnimalCluster);
}
```

somSupportsMethod

This method returns a boolean indicating whether instances of a class respond to a given (static or dynamic) method.

For backward compatibility, this method does *not* take an **Environment** parameter.

Syntax

```
boolean somSupportsMethod   (SOMClass  receiver, somId  methodId)
```

Parameters

receiver (SOMClass)
A pointer to the class object to be tested.

methodId (somId)
An ID that represents the name of the method.

Returns

rc (boolean)
Returns 1 (true) if instances of the specified class support the specified method, and 0 (false) otherwise.

Remarks

The **somSupportsMethod** method determines if instances of the specified class respond to the specified (static or dynamic) method.

Original Class

SOMClass

Related Methods

Methods

- **somRespondsTo**

Example Code

```
/* ------------------------------------------------
   Note:  animal supports a setSound method;
          animal does not support a doTrick method.
   ------------------------------------------------ */
#include <animal.h>
main()
{
  SOMClass animalClass;
  char *methodName1 = "setSound";
  char *methodName2 = "doTrick";
  animalClass =
      AnimalNewClass(Animal_MajorVersion, Animal_MinorVersion);
  if (_somSupportsMethod(animalClass,
                         somIdFromString(methodName1)))
      somPrintf("Animals respond to %s\n", methodName1);
  if (_somSupportsMethod(animalClass,
                          somIdFromString(methodName2)))
      somPrintf("Animals respond to %s\n", methodName2);
}

/*
Output from this program:
Animals respond to setSound
*/
```

SOMClassMgr

File stem: somcm

Base

SOMObject

Metaclass

SOMClass

Ancestor Classes

SOMObject

Description

One instance of **SOMClassMgr** is created automatically during SOM initialization. This instance (pointed to by the global variable, **SOMClassMgrObject**) acts as a run-time registry for all SOM class objects that exist within the current process and assists in the dynamic loading and unloading of class libraries.

You can subclass **SOMClassMgr** to augment the functionality of its registry. To have an instance of your subclass replace the SOM-supplied **SOMClassMgrObject**, use the **somMergeInto** method to place the existing registry information from **SOMClassMgrObject** into your new class-manager object.

Types

```
interface Repository;
SOMClass *SOMClassArray;
```

Attributes

Listed below is each available attribute with its corresponding type in parentheses, followed by a description of its purpose.

somInterfaceRepository (Repository)

The SOM Interface Repository object. If the Interface Repository is not available or cannot be initialized, this attribute returns NULL. When your program finishes using the Repository object, it should call the **somDestruct** method to release the reference, using a non-zero value for the *doFree* parameter.

somRegisteredClasses (sequence<SOMClass>)

This is a "readonly" attribute that returns a sequence containing all of the class objects registered in the current process. When you have finished using the returned sequence, you should free the sequence's buffer using **SOMFree**. Here is a fragment of code written in C that illustrates the proper use of this attribute:

```
sequence(SOMClass) clsList;

clsList = SOMClassMgr__get_somRegisteredClasses (SOMClassMgrObject);
somPrintf ("Currently registered classes:\n");
for (i=0; i<clsList._length; i++)
   somPrintf ("\t%s\n", SOMClass_somGetName (clsList._buffer[i]));
SOMFree (clsList._buffer);
```

New Methods

The following list shows all the SOMClassMgr methods.

Group: Basic Functions:

- somLoadClassFile
- somLocateClassFile
- somRegisterClass
- somUnloadClassFile
- somUnregisterClass

Group: Access:

- somGetInitFunction
- somGetRelatedClasses

Group: Dynamic:

- somClassFormId
- somFindClass
- somFindClsInFile
- somMergeInto
- somSubstituteClass

Overridden Methods

The following list shows all the methods overridden by the SOMClassMgr class. These methods are overridden in order to modify the behavior defined by an ancestor class.

- somDumpSelfInt
- somInit
- somUninit

somClassFromId

This method finds a class object, given its somId, if it already exists. Does not load the class.

For backward compatibility, this method does *not* take an **Environment** parameter.

Syntax

```
SOMClass somClassFromId  (SOMClassMgr  receiver, somId  classId)
```

Parameters

receiver (SOMClassMgr)
Usually **SOMClassMgrObject** (or a pointer to an instance of a user-supplied subclass of **SOMClassMgr**).

classId (somId)
The somId of the class. This can be obtained from the name of the class using the **somIdFromString** function.

Returns

rc (SOMClass)
Returns a pointer to the class, or NULL if the class object does not yet exist.

Remarks

Finds a class object, given its somId, if it already exists. Does not load the class.

Use the **somClassFromId** method instead of **somFindClass** when you do *not* want the class to be automatically loaded if it does not already exist in the current process.

Original Class

SOMClassMgr

Related Methods

Methods

- **somFindClass**
- **somFindClsInFile**

Example Code

```
#include <som.h>

main () {
    SOMClass myClass;
    char *myClassName = "Animal";
    somId animalId;

    somEnvironmentNew ();
    animalId = somIdFromString (myClassName);
    myClass = SOMClassMgr_somClassFromId (SOMClassMgrObject,
                                                        animalId);
    if (!myClass)
        somPrintf ("Class %s has not been loaded.\n", myClassName);
    SOMFree (animalId);
        }
```

This program produces the following output:

```
Class Animal has not yet been loaded.
```

somFindClass

This method finds the class object for a class.

For backward compatibility, this method does *not* take an **Environment** parameter.

Syntax

```
SOMClass somFindClass  (SOMClassMgr  receiver, somId  classId,
                        long  majorVersion, long  minorVersion)
```

Parameters

receiver (SOMClassMgr)
Usually **SOMClassMgrObject** (or a pointer to an instance of a user-supplied subclass of **SOMClassMgr**).

classId (somId)
The **somId** representing the name of the class.

majorVersion (long)
The class's major version number.

minorVersion (long)
The class's minor version number.

Returns

rc (SOMClass)
A pointer to the requested class object, or NULL if the class could not be found or created.

Remarks

The **somFindClass** method returns the class object for the specified class. This method first uses **somLocateClassFile** (see paragraph below) to obtain the name of the file where the class's code resides, then uses **somFindClsInFile**.

If the requested class has not yet been created, the **somFindClass** method attempts to load the class dynamically by loading its dynamically linked library and invoking its "new class" procedure.

The **somLocateClassFile** method uses the following steps:

1. If the entry in the Interface Repository for the class specified by *classId* contains a **dllname** modifier, this value is used as the file name for loading the library. (For information about the **dllname** modifier, refer to the topic "Modifier statements" in Chapter 4, "SOM IDL and the SOM Compiler," of the *SOM Programming Guide*.
2. In the absence of a **dllname** modifier, the class name is assumed to be the file name for the library. Use the **somFindClsInFile** method if you wish to explicitly pass the file name as an argument.

If *majorVersion* and *minorVersion* are not both zero, they are used to check the class version information against the caller's expectations. An implementation is compatible with the specified version numbers if it has the same major version number and a minor version number that is equal to or greater than *minorVersion*.

Original Class

SOMClassMgr

Related Methods

Methods

- **somFindClsInFile**
- **somLocateClassFile**

Example Code

```
#include <som.h>

/*
 *  This program creates a class object
 *  (from a DLL) without requiring the
 *  usage binding file (.h or .xh) for
 *  the class.
 */

void main ()
{
    SOMClass myClass;
    somId animalId;

    somEnvironmentNew ();
    animalId = somIdFromString ("Animal");

/*  The next statement is equivalent to:
 *    #include "animal.h"
 *    myClass = AnimalNewClass (0, 0);
 */
    myClass = SOMClassMgr_somFindClass (SOMClassMgrObject,
                                                  animalId, 0, 0);
    if (myClass)
        somPrintf ("myClass: %s\n", SOMClass_somGetName (myClass));
    else
        somPrintf ("Class %s could not be dynamically loaded\n",
                                        somStringFromId (animalId));
    SOMFree (animalId);
}
```

somFindClsInFile

This method finds a class object for a class.

For backward compatibility, this method does *not* take an **Environment** parameter.

Syntax

```
SOMClass somFindClsInFile  (SOMClassMgr receiver, somId classId,
                            long majorVersion, long minorVersion,
                            string file)
```

Parameters

receiver (SOMClassMgr)
Usually **SOMClassMgrObject** (or a pointer to an instance of a user-supplied subclass of **SOMClassMgr**).

classId (somId)
The somId representing the name of the class.

majorVersion (long)
The class's major version number.

minorVersion (long)
The class's minor version number.

file (string)
A string representing the filename to be used if dynamic loading is required.

Returns

rc (SOMClass)
A pointer to the requested class object, or NULL if the class could not be found or created.

Remarks

The **somFindClsInFile** method returns the class object for the specified class. This method is the same as **somFindClass** except that the caller provides the filename to be used if dynamic loading is needed.

somFindClsInFile

If the requested class has not yet been created the **somFindClsInFile** method attempts to load the class dynamically by loading the specified library and invoking its "new class" procedure.

If *majorVersion* and *minorVersion* are not both zero, they are used to check the class version information against the caller's expectations. An implementation is compatible with the specified version numbers if it has the same major version number and a minor version number that is equal to or greater than *minorVersion*.

Original Class

SOMClassMgr

Related Methods

Methods

- **somFindClass**

Example Code

```
#include <som.h>
 /*                                          */
 /*  This program loads a class and creates  */
 /*  an instance of it without requiring the */
 /*  binding (.h) file for the class.        */
 /*                                          */
void main()
{
   SOMObject myAnimal;
   SOMClass animalClass;
   char *animalName = "Animal";

        /*
         * Filenames will be different for AIX and OS/2
         * Set animalfile to "C:\\MYDLLS\\ANIMAL.DLL" for OS/2.
         * Set animalfile to "/mydlls/animal.dll" for AIX.
         */

   char *animalFile = "/mydlls/animal.dll";     /* AIX filename */

   somEnvironmentNew();
   animalClass = _somFindClsInFile (SOMClassMgrObject,
                                    somIdFromString(animalName),
                                    0, 0,
                                    animalFile);
   myAnimal = _somNew (animalClass);
   somPrintf("The class of myAnimal is %s.\.n",
       _somGetClassName(myAnimal));
   _somFree(myAnimal);
}
   /*   Output from this program:  */
   /*   The class of myAnimal is Animal. */
```

somGetInitFunction

This method obtains the name of the function that initializes the SOM classes in a class library.

For backward compatibility, this method does *not* take an **Environment** parameter.

Syntax

```
string somGetInitFunction   (SOMClassMgr  receiver)
```

Parameters

receiver (SOMClassMgr)
Usually **SOMClassMgrObject** (or a pointer to an instance of a user-supplied subclass of **SOMClassMgr**).

Returns

rc (string)
Returns a string that names the initialization function of class libraries. By default, this name is the value of the global variable **SOMClassInitFuncName**, the default value of which is **SOMInitModule**.

Remarks

The **somGetInitFunction** method supplies the name of the initialization function for OS/2 class libraries (DLLs) that contain more than one SOM class. The default implementation returns the value of the global variable **SOMClassInitFuncName**, which by default is set to the value "SOMInitModule".

For AIX, the name of the class initialization function is not important, since AIX class libraries should always be constructed as shared libraries with a designated entry point which can be executed automatically by the loader when the class is loaded. Consequently, the result of this method is not significant on AIX.

Similarly, if an OS/2 class library (DLL) has been constructed with a DLL initialization function assigned by the linker, you can choose to invoke the **<className>NewClass** functions for all of the classes in the DLL during DLL initialization. In this case (as on AIX), there is no need to export a "SOMInitModule" function. On the other hand, if your compiler does not provide a convenient mechanism for creating a DLL initialization function, you can elect to export a

function named "SOMInitModule" (or whatever name is ultimately returned by the **somGetInitFunction** method).

The OS/2 **SOMClassMgrObject**, after loading a class library, will invoke the method **somGetInitFunction** to obtain the name of a possible initialization function. If this name has been exported by the class library just loaded, the **SOMClassMgrObject** calls this function to initialize the classes in the library. If the name has not been exported by the DLL, the **SOMClassMgrObject** then looks for an exported name of the form **<className>NewClass**, where <className> is the name of the class supplied with the method that caused the DLL to be loaded. If the DLL exports this name, it is invoked to create the named class.

On Windows, the SOM class manager does *not* call **SOMInitModule**. It must be called from the default Windows DLL initialization function, LibMain. This call is made indirectly through the **SOM_ClassLibrary** macro.

Regardless of the technique employed, the **SOMClassMgrObject** expects that all classes packaged in a single class library will be created during this sequence.

This method is generally not invoked directly by users. User-defined subclasses of **SOMClassMgr**, however, can override this method.

Original Class

SOMClassMgr

Related Methods

Methods

- **somFindClass**
- **somFindClsInFile**

Functions

- **SOMInitModule**

Macros

- **SOM_ClassLibrary**

somGetRelatedClasses

This method returns an array of class objects that were all registered during the dynamic loading of a class.

For backward compatibility, this method does *not* take an **Environment** parameter.

Syntax

```
SOMClass *somGetRelatedClasses  (SOMClassMgr  receiver,
                                 SOMClass  classObj)
```

Parameters

receiver (SOMClassMgr)
Usually a pointer to **SOMClassMgrObject**, or a pointer to an instance of a user-defined subclass of **SOMClassMgr**.

classObj (SOMClass)
A pointer to a **SOMClass** object.

Returns

rc (SOMClass *)
Returns a pointer to an array of pointers to class objects, or NULL, if the specified class was not dynamically loaded.

Remarks

The **somGetRelatedClasses** method returns an array of class objects that were all registered during the dynamic loading of the specified class. These classes are considered to define an affinity group. Any class is a member of at most one affinity group. The affinity group returned by this call is the one containing the class identified by the *classObj* parameter.

The first element in the array is either the class that caused the group to be loaded, or the special value -1, which means that the class manager is currently in the process of unregistering and deleting the affinity group (only class-manager objects would ever see this value). The remainder of the array consists of pointers to class objects, ordered in reverse chronological sequence to that in which they were originally registered. This list includes the given argument, *classObj*, as one of its elements, as

well as the class that caused the group to be loaded (also given by the first element of the array). The array is terminated by a NULL pointer as the last element.

Use **SOMFree** to release the array when it is no longer needed. If the supplied class was not dynamically loaded, it is not a member of any affinity group and NULL is returned.

Original Class

SOMClassMgr

Related Methods

Methods

- **somGetInitFunction**

Example Code

```
#include <som.h>
SOMClass myClass, *relatedClasses;
string className;
long i;

className = SOMClass_somGetName (myClass));
relatedClasses = SOMClassMgr_somGetRelatedClasses
                                (SOMClassMgrObject, myClass);
if (relatedClasses && *relatedClasses) {
    somPrintf ("Class=%s, related classes are: ", className);
    for (i=1; relatedClasses[i]; i++)
        somPrintf ("%s ",SOMClass_somGetName (relatedClasses[i]));
    somPrintf ("\n");
    somPrintf ("Class that caused loading was %s\n",
        relatedClasses[0] == (SOMClass) -1 ? "-1" :
            SOMClass_somGetName (relatedClasses[0]));
    SOMFree (relatedClasses);
} else
    somPrintf ("No classes related to %s\n", className);
```

somLoadClassFile

This method dynamically loads a class.

For backward compatibility, this method does *not* take an **Environment** parameter.

Syntax

```
SOMClass somLoadClassFile (SOMClassMgr receiver, somID classId,
                           long majorVersion, long minorVersion,
                           string file)
```

Parameters

receiver (SOMClassMgr)
Usually **SOMClassMgrObject** (or a pointer to an instance of a user-supplied subclass of **SOMClassMgr**).

classId (somID)
The somId representing the name of the class to load.

majorVersion (long)
The major version number used to check the compatibility of the class's implementation with the caller's expectations.

minorVersion (long)
The minor version number used to check the compatibility of the class's implementation with the caller's expectations.

file (string)
The name of the dynamically linked library file containing the class. The name can be either a simple, unqualified name (without any extension) or a fully qualified (or path) file name, as appropriate for your operating system. For example, on OS/2, *file* could be `c:\myhome\myapp\basename.dll` or else `basename` (but not `basename.dll`).

Returns

rc (SOMClass)
Returns a pointer to the class object, or NULL if the class could not be loaded or the class object could not be created.

Remarks

The **SOMClassMgr** object uses the **somLoadClassFile** method to load a class dynamically during the execution of **somFindClass** or **somFindClsInFile**. A SOM class object representing the class is expected to be created and registered as a result of this method.

The **somLoadClassFile** method can be overridden to load or create classes dynamically using your own mechanisms. If you simply wish to change the name of the procedure that is called to initialize the classes in a library, override **somGetInitFunction** instead.

This method is provided to permit user-created subclasses of **SOMClassMgr** to handle the loading of classes by overriding this method. Do not invoke this method directly; instead, use **somFindClass** or **somFindClsInFile**.

Original Class

SOMClassMgr

Related Methods

Methods

- **somFindClass**
- **somFindClsInFile**
- **somGetInitFunction**
- **somUnloadClassFile**

somLocateClassFile

This method determines the file that holds a class to be dynamically loaded.

For backward compatibility, this method does *not* take an **Environment** parameter.

Syntax

```
string somLocateClassFile  (SOMClassMgr  receiver, somId  classId,
                            long  majorVersion, long  minorVersion)
```

Parameters

receiver (SOMClassMgr)
Usually **SOMClassMgrObject** (or a pointer to an instance of a user-supplied subclass of **SOMClassMgr**).

classId (somId)
The somId representing the name of the class to locate.

majorVersion (long)
The major version number used to check the compatibility of the class's implementation with the caller's expectations.

minorVersion (long)
The minor version number used to check the compatibility of the class's implementation with the caller's expectations.

Returns

rc (string)
Returns the name of the file containing the class.

Remarks

The **SOMClassMgr** object uses the **somLocateClassFile** method when executing **somFindclass** to obtain the name of a file to use when dynamically loading a class. The default implementation consults the Interface Repository for the value of the *dllname* modifier of the class; if no *dllname* modifier was specified, the method simply returns the class name as the expected filename.

If you override the **somLocateClassFile** method in a user-supplied subclass of **SOMClassMgr,** the name you return can be either a simple, unqualified name

without any extension or a fully-qualified file name. Generally speaking, you would not invoke this method directly. It is provided to permit customization of subclasses of **SOMClassMgr** through overriding.

Original Class

SOMClassMgr

Related Methods

Methods

- **somFindClass**
- **somFindClsInFile**
- **somGetInitFunction**
- **somLoadClassFile**
- **somUnloadClassFile**

somMergeInto

This method transfers SOM class registry information to another **SOMClassMgr** instance.

Note: For backward compatibility, this method does *not* take an **Environment** parameter.

Syntax

```
void somMergeInto  (SOMClassMgr  receiver, SOMClassMgr  target)
```

Parameters

receiver (SOMClassMgr)
Usually **SOMClassMgrObject** (or a pointer to an instance of a user-supplied subclass of **SOMClassMgr**).

target (SOMClassMgr)
A pointer to another instance of **SOMClassMgr** or one of its subclasses.

Returns

rc (void)

Remarks

The **somMergeInto** method transfers the **SOMClassMgr** registry information from one object to another. The target object is required to be an instance of **SOMClassMgr** or one of its subclasses. At the completion of this operation, the target object can function as a replacement for the receiver. The receiver object (which is then in a newly uninitialized state) is placed in a mode where all methods invoked on it will be delegated to the target object. If the receiving object is the instance pointed to by the global variable **SOMClassMgrObject**, then **SOMClassMgrObject** is reassigned to point to the target object.

Subclasses of **SOMClassMgr** that override the **somMergeInto** method should transfer their section of the class manager object from the target to the receiver, then invoke their parent's **somMergeInto** method as the final step.

Invoke this method only if you are creating your own subclass of **SOMClassMgr**. You can invoke **somMergeInto** from an initializer for your new class manager.

Original Class

SOMClassMgr

Example Code

```
// === IDL For the New Class Manager ===

#include <somcm.idl>

interface NewCM  :  SOMClassMgr {
        implementation   {
                somDefaultInit:  override;
       };
};

// === C++ implementation for NewCM ===

#define SOM_Module_merge_Source
#include "merge.xih"
SOM_Scope void SOMLINK somDefaultInit(NewCM *somSelf,
                                                        somInitCtrl* ctrl)
{
        NewCMData *somThis;   /* set in BeginInitializer */
        somInitCtrl globalCtrl;
        somBooleanVector myMask;
        NewCMMethodDebug ("NewCM","somDefaultInit");
        NewCM_BeginInitializer_somDefaultInit;

        NewC_Init_SOMClassMgr_somDefaultInit(somSelf, ctrl);

        /*
         *  local NewCM initialization code added by programmer
         */

        SOMClassMgrObject->somMergeInto(somSelf);
}
```

somMergeInto

```
// === C++ test program ===

#include <merge.xh>
main()
{
        NewCM *ncm = new NewCM;
        SOMClassMgrObject->somDumpSelf(0);
}

// === Output from test program ===

{An instance of class NewCM at address 20084388
 1 classIdSpaceSize:  3200
 1 classIdHashTableSize:  397
 1 loadAffinity:  0
 1 nextLoadAffinity:  1
 1 IR Class:  00000000,  IR Object:  00000000
 1          -Class-- -Token-- Aff Seq ---Id--- Name
 1 ffl    0" 20077A48 00000000 000 001 2008260C SOMObject
 1 ffl    1" 2007FB38 00000000 000 000 200825EC SOMClassMgr
 1 ffl    2" 20083B08 00000000 000 004 2008436C NewCM
 1 ffl    3" 20077BD8 00000000 000 002 2008262C SOMClass
 1 ffl    4" 20082668 00000000 000 003 2008315C
 SOMParentDerivedMetaclass
}
```

somRegisterClass

This method adds a class object to the SOM run-time class registry.

Note: For backward compatibility, this method does *not* take an **Environment** parameter.

Syntax

```
void somRegisterClass  (SOMClassMgr  receiver, SOMClass  classObj)
```

Parameters

receiver (SOMClassMgr)
Usually **SOMClassMgrObject** (or a pointer to an instance of a user-supplied subclass of **SOMClassMgr**).

classObj (SOMClass)
A pointer to the class object to add to the SOM class registry.

Returns

rc (void)

Remarks

The **somRegisterClass** method adds a class object to the SOM run-time class registry maintained by **SOMClassMgrObject**.

All SOM run-time class objects should be registered with the **SOMClassMgrObject**. This is done automatically during the execution of the **somClassReady** method as class objects are created.

Original Class

SOMClassMgr

Related Methods

Methods:

- **somUnregisterClass**

somSubstituteClass

This method causes the **somFindClass**, **somFindClsInFile**, and **somClassFromId** methods to substitute one class for another.

Note: For backward compatibility, this method does *not* take an **Environment** parameter.

Syntax

```
long somSubstituteClass  (SOMClassMgr  receiver, string  origClassName,
                          string  newClassName)
```

Parameters

receiver (SOMClassMgr)
Usually **SOMClassMgrObject** or a pointer to an instance of a user-defined subclass of **SOMClassMgr**.

origClassName (string)
A NULL terminated string containing the old class name.

newClassName (string)
A NULL terminated string containing the new class name.

Returns

rc (long)
The **somSubstituteClass** method returns a value of zero to indicate success; a non-zero value indicates an error was detected.

Remarks

The **somSubstituteClass** method causes the **somFindClass**, **somFindClsInFile**, and **somClassFromId** methods to return the class named *newClassName* whenever they would normally return the class named *origClassName*. This effectively results in class *newClassName* replacing or substituting for class *origClassName*. For example, the **<origClassName>New** macro will subsequently create instances of *newClassName*.

Some restrictions are enforced to ensure that this works well. Both class *origClassName* and class *newClassName* must have been already registered before

issuing this method, and *newClass* must be an immediate child of *origClass*. In addition (although not enforced), no instances should exist of either class at the time this method is invoked.

A convenience macro (**SOM_SubstituteClass**) is provided for C or C++ users. In one operation, it creates both the old and the new class and then substitutes the new one in place of the old. The use of both the **somSubstituteClass** method and the **SOM_SubstituteClass** macro is illustrated in the example below.

Original Class

SOMClassMgr

Related Methods

Methods:

- **somClassFromId**
- **somFindClass**
- **somFindClsInFile**
- **somMergeInto**

Example Code

```
#include <student.h>
#include <mystud.h>

/* Macro form */
SOM_SubstituteClass (Student, MyStudent);
/* Direct use of the method, equivalent to
 * the macro form above.
 */
{
SOMClass origClass, replacementClass;

origClass = StudentNewClass (Student_MajorVersion,
                                    Student_MinorVersion);
replacementClass = MyStudentNewClass (MyStudent_MajorVersion,
                                     MyStudent_MinorVersion);
SOMClassMgr_somSubstituteClass (
    SOMClass_somGetName (origClass),
    SOMClass_somGetName (replacementClass));
}
```

somUnloadClassFile

This method unloads a dynamically loaded class and frees the class's object.

Note: For backward compatibility, this method does *not* take an **Environment** parameter.

Syntax

```
long somUnloadClassFile   (SOMClassMgr  receiver, SOMClass  class)
```

Parameters

receiver (SOMClassMgr)
Usually **SOMClassMgrObject** (or a pointer to an instance of a user-supplied subclass of **SOMClassMgr**).

class (SOMClass)
A pointer to the class to be unloaded.

Returns

rc (long)
The **somUnloadClassFile** method returns 0 if the class was successfully unloaded; otherwise, it returns a system-specific non-zero error code from either the OS/2 **DosFreeModule** or the AIX **unload** system call.

Remarks

The **somUnregisterClass** method uses the **somUnloadClassFile** method to unload a dynamically loaded class. This releases the class's code and unregisters all classes in the same affinity group. (Use **somGetRelatedClasses** to find out which other classes are in the same affinity group.)

The class object is freed whether or not the class' s shared library could be unloaded. If the class was not registered, an error condition is raised and **SOMError** is invoked. This method is provided to permit user-created subclasses of **SOMClassMgr** to handle the unloading of classes by overriding this method. Do not invoke this method directly; instead, invoke **somUnregisterClass**.

Original Class

SOMClassMgr

Related Methods

Methods:

- **somLoadClassFile**
- **somRegisterClass**
- **somUnregisterClass**
- **somGetRelatedClasses**

somUnregisterClass

This method removes a class object from the SOM run-time class registry.

Note: For backward compatibility, this method does *not* take an **Environment** parameter.

Syntax

```
long somUnregisterClass  (SOMClassMgr  receiver, SOMClass  class)
```

Parameters

receiver (SOMClassMgr)
Usually **SOMClassMgrObject** (or a pointer to an instance of a user-supplied subclass of **SOMClassMgr**).

class (SOMClass)
A pointer to the class to be unregistered.

Returns

rc (long)
The **somUnregisterClass** method returns 0 for a successful completion, or non-zero to denote failure.

Remarks

The **somUnregisterClass** method unregisters a SOM class and frees the class object. If the class was dynamically loaded, it is also unloaded using **somUnloadClassFile** (which causes its entire affinity group to be unloaded as well).

Original Class

SOMClassMgr

Related Methods

Methods:

- **somLoadClassFile**
- **somRegisterClass**
- **somUnloadClassFile**

Example Code

```
#include <som.h>

void main ()
{
    long rc;  /* Return code */
    SOMClass animalClass;

    /* The next 2 lines declare a static form of somId */
    string animalClassName = "Animal";
    somId animalId = &animalClassName;

    somEnvironmentNew ();
    animalClass = SOMClassMgr_somFindClass (SOMClassMgrObject,
                                            animalId, 0, 0);
    if (!animalClass) {
       somPrintf ("Could not load class.\n");
       return;
    }
    rc = SOMClassMgr_somUnregisterClass (SOMClassMgrObject,
                                            animalClass);
    if (rc)
        somPrintf ("Could not unregister class, error code: %ld.\n",
                                                            rc);
    else
        somPrintf ("Class successfully unloaded.\n");
}
```

SOMObject

File stem: somobj

Base

None.

Metaclass

SOMClass

Ancestor Classes

None

Description

SOMObject is the root class for all SOM classes. That is, all SOM classes must be subclasses of **SOMObject** or of some other class derived from **SOMObject**. **SOMObject** introduces no instance data, so objects whose classes inherit from **SOMObject** incur no size increase. They do inherit a suite of methods that provide the behavior required of all SOM objects. Three of these methods are typically overridden by any subclass that has instance data— **somDefaultInit**, **somDestruct**, and **somDumpSelfInt**. See the descriptions of these methods for further information.

New Methods

The following list shows all the SOMObject methods.

Group: Initialization/Termination:

- somFree
- somDefaultInit
- somDestruct
- somInit
- somUninit
- somDefaultAssign
- somDefaultConstAssign
- somDefaultConstCopyInit
- somDefaultCopyInit

Group: Access:

- somGetClass
- somGetClassName
- somGetSize

Group: Testing:

- somIsA
- somIsInstanceOf
- somRespondsTo

Group: Dynamic:

- somDispatchA
- somDispatchD
- somDispatchL
- somDispatchV
- somDispatch
- somClassDispatch
- somCastObj
- somResetObj

Group: Developer Support:

- somDumpSelf
- somDumpSelfInt
- somPrintSelf

Overridden Methods

There are currently no overridden methods defined for the SOMObject class.

somCastObj

This method changes the behavior of an object to that defined by any ancestor of the true class of the object.

Syntax

```
boolean somCastObj   (SOMObject  receiver, SOMClass  ancestor)
```

Parameters

receiver (SOMObject)
A pointer to an object of type **SOMObject**.

ancestor (SOMClass)
A pointer to a class that is an ancestor of the actual class of the receiver.

Returns

rc (boolean)

TRUE	Returns 1 (TRUE) if the operation is successful.
FALSE	Returns 0 (false) if the operation is not successful. The operation fails if ancestor is not actually an ancestor of the class of the object.

Remarks

This method changes the behavior of an object so that its behavior will be that of an instance of the indicated ancestor class (with respect to any method supported by the ancestor). The behavior of the object on methods not supported by the ancestor remains unchanged.

This operation actually changes the class of the object (since an object's behavior is defined by its class). The name of the new class is derived from the initial name of the object's class and the name of the ancestor class, as illustrated in the Example.

This method may be used on an object multiple times, always with the restriction that the ancestor class whose behavior is selected is actually an ancestor of the true (original) class of the object.

Original Class

SOMObject

Related Methods

Methods

- **somResetObj**

Example Code

```
#include <som.h>
main()
{
   SOMClassMgr cm = somEnvironmentNew();
   SOM_Test(1 == _somCastObj(cm, _SOMObject));
   _somDumpSelf(cm, 0));
   SOM_Test(1 == _somResetObj(cm));
   _somDumpSelf(cm, 0);
}

/* output:
 *  {An instance of class SOMClassMgr->SOMObject
 *   at address 20061268
 *  }
 *  {An instance of class SOMClassMgr at address 20061268
 *   ... <SOMClassMgr State Information> ...
 *  }
 */
```

somDefaultAssign

This method provides support for an object-assignment operator. May be overridden, but, if appropriate, **somDefaultConstAssign** should be overridden instead.

Syntax

```
void somDefaultAssign  (SOMObject  receiver, somInitCtrl  ctrl,
                        SOMObject  fromObj)
```

Parameters

receiver (SOMObject)
A pointer to an object of an arbitrary SOM class, *S*.

ctrl (somInitCtrl)
A pointer to a **somInitCtrl** structure, or NULL.

fromObj (SOMObject)
A pointer to an object of class *S* or some class descended from *S*.

Returns

rc (void)

Remarks

In C++, assignment to an object of class "X" is accomplished by using (an appropriate overloading of) the assignment operator provided by "X". To make assignment available on all SOM objects, **SOMObject** provides the **somDefaultAssign** and **somDefaultConstAssign** methods. The default behavior of these methods is that they do a shallow copy of data from one object to another. Users should generally use the **somDefaultAssign** method for doing object assignment.

When a shallow copy is not appropriate for the data introduced by a class, and it is possible to perform the copy without modifying *fromObj*, it is recommended that the class implementor override the **somDefaultConstAssign** method for that class.

The considerations important to overriding **somDefaultAssign** are similar to those described in the *SOM Programming Guide* for overriding **somDefaultInit**. (See

"Initializing and Uninitializing Objects" in Chapter 5, "Implementing Classes in SOM.") The basic difference between **somDefaultInit** and **somDefaultAssign** is that the latter method takes an object (*fromObj*) as a source argument for assignment of values to the receiver.

Original Class

SOMObject

Related Methods

Methods

- **somDefaultInit**
- **somDefaultConstAssign**
- **somDefaultCopyInit**
- **somDefaultConstCopyInit**

Example Code

```
// C++ SOMObjects Toolkit Code
#include <Y.xh>

main()
{
        X *x = new X;
        Y *y = new Y;  // assume Y is derived from X
        x->somDefaultAssign(0,y)
        //  the x object has now been assigned values from y
}
```

somDefaultConstAssign

Provides support for a "const" object-assignment operator. Designed to be overridden.

Syntax

```
void somDefaultConstAssign  (SOMObject  receiver, somInitCtrl  ctrl,
                             SOMObject  fromObj)
```

Parameters

receiver (SOMObject)
A pointer to an object of an arbitrary SOM class, *S*.

ctrl (somInitCtrl)
A pointer to a **somInitCtrl** structure, or NULL.

fromObj (SOMObject)
A pointer to an object of class *S* or some class descended from *S*.

Returns

rc (void)

Remarks

In C++, assignment to an object of class "X" is accomplished by using (an appropriate overloading of) the assignment operator provided by "X". To make assignment available on all SOM objects, **SOMObject** introduces the **somDefaultAssign** and **somDefaultConstAssign** methods. The default behavior of these methods is to perform a shallow copy of data from one object to another. When this default is not appropriate for a class, and it is possible to perform the copy without modifying *fromObj*, it is recommended that the class implementor override the **somDefaultConstAssign** method.

Generally, an object user should use the **somDefaultAssign** method to perform object assignment.

The considerations important to overriding **somDefaultConstAssign** are similar to those described in the *SOM Programming Guide* for overriding **somDefaultInit**. (See

"Initializing and Uninitializing Objects" in Chapter 5, "Implementing Classes in SOM.") The basic difference between **somDefaultInit** and **somDefaultConstAssign** is that the latter method takes an object (*fromObj*) as an argument that is to be copied.

Original Class

SOMObject

Related Methods

Methods

- **somDefaultInit**
- **somDefaultAssign**
- **somDefaultCopyInit**
- **somDefaultConstCopyInit**

Example Code

```
// IDL for a class that overrides somDefaultConstAssign
#include <X.idl>

interface Y : X  {
        implementation  {
                somDefaultConstAssign: override, init;
        };
};
```

somDefaultConstCopyInit

This method provides support for passing objects as call-by-value object parameters in methods introduced by DTS C++ classes. Designed to be overridden.

Syntax

```
void somDefaultConstCopyInit  (SOMObject  receiver, somInitCtrl  ctrl,
                               SOMObject  fromObj)
```

Parameters

receiver (SOMObject)
A pointer to an uninitialized object of an arbitrary SOM class, *S*.

ctrl (somInitCtrl)
A pointer to a **somInitCtrl** structure, or NULL.

fromObj (SOMObject)
A pointer to an object of class *S* or some class descended from *S*.

Returns

rc (void)

Remarks

The **somDefaultConstCopyInit** method would be called a "copy constructor" in C++. In SOM, this concept is supported using an object initializer that accepts the object to be copied as an argument. Copy constructors are used in C++ to pass objects by value. They initialize one object by making it be a copy of another object. In SOM, objects are always passed by reference, so arguments to DTS C++ methods that receive call-by-value object parameters are actually passed by reference. But, to correctly support the semantics of DTS C++ call-by-value arguments, it is necessary to actually pass a *copy* of the intended argument. A copy constructor can be used to make this copy.

The default behavior provided by **somDefaultConstCopyInit** is to do a shallow copy of each ancestor class's introduced instance variables. The object being copied is not changed. When a shallow copy is not appropriate, and it is possible to avoid changing *fromObj*, a class implementor should override **somDefaultConstCopyInit** (for

example, to do a deep copy for certain variables), but should respect the constraint of not modifying the object being copied.

In general, object users should use **somDefaultCopyInit** to copy an object.

The considerations important to overriding **somDefaultConstCopyInit** are similar to those described in the *SOM Programming Guide* for overriding **somDefaultInit**. (See "Initializing and Uninitializing Objects" in Chapter 5, "Implementing Classes in SOM.") The basic difference between **somDefaultInit** and **somDefaultConstCopyInit** is that the latter method takes an object (*fromObj*) as an argument that is to be copied.

Original Class

SOMObject

Related Methods

Methods

- **somDefaultInit**
- **somDefaultCopyInit**
- **somDefaultAssign**
- **somDefaultConstAssign**

Example Code

```
// IDL for a class that overrides somDefaultConstCopyInit
interface X : SOMObject
{
        implementation {
                somDefaultConstCopyInit: override, init;
        };
};
```

somDefaultCopyInit

This method provides support for call-by-value object parameters in methods introduced by DTS C++ classes. May be overridden, but, if appropriate, **somDefaultConstCopyInit** should be overridden instead.

Syntax

```
void somDefaultCopyInit  (SOMObject  receiver, somInitCtrl  ctrl,
                          SOMObject  fromObj)
```

Parameters

receiver (SOMObject)
A pointer to an uninitialized object of an arbitrary SOM class, *S*.

ctrl (somInitCtrl)
A pointer to a **somInitCtrl** structure, or NULL.

fromObj (SOMObject)
A pointer to an object of class *S* or some class descended from *S*.

Returns

rc (void)

Remarks

The **somDefaultCopyInit** method would be called a "copy constructor" in C++. In SOM, this concept is supported using an object initializer that accepts the object to be copied as an argument. Copy constructors are used in C++ to pass objects by value. They initialize one object by making it be a copy of another object. In SOM, objects are always passed by reference, so arguments to DTS C++ methods that receive call-by-value object parameters are actually passed by reference. But, to correctly support the semantics of DTS C++ call-by-value arguments, it is necessary to actually pass a *copy* of the intended argument. In general, **somDefaultCopyInit** should be used to make this copy.

The default behavior provided by **somDefaultCopyInit** is to do a shallow copy of each ancestor class's introduced instance variables. However, a class may always override this default behavior (for example, to do a deep copy for certain variables).

If it is possible to avoid modification of fromObj when doing the copy, the method **somDefaultConstCopyInit** should be overridden for this purpose. Only if this is not possible (and shallow copy is not appropriate) would it be appropriate to override **somDefaultCopyInit**.

The considerations important to overriding **somDefaultCopyInit** are similar to those described in the *SOM Programming Guide* for overriding **somDefaultInit**. (See "Initializing and Uninitializing Objects" in Chapter 5, "Implementing Classes in SOM.") The basic difference between **somDefaultInit** and **somDefaultCopyInit** is that the latter method takes an object (*fromObj*) as an argument that is to be copied.

Original Class

SOMObject

Related Methods

Methods

- **somDefaultInit**
- **somDefaultConstCopyInit**
- **somDefaultAssign**
- **somDefaultConstAssign**

Example Code

```
// IDL produced by a DTS C++ compiler for a DTS C++ class
interface X : SOMObject
{
        void foo(in SOMClass arg);
        implementation {
                foo: cxxdecl = "void foo(SOMClass arg)";  //  !! call-by-value
        };
};

// C++ SOMObjects Toolkit Code
#include <X.xh>
#include <somcls.xh>
main()
{
        X *x = new X;
        SOMClass *arg = _SOMClass->somNewNoInit();
        // make arg be a copy of the X class object
        arg->somDefaultCopyInit(0,_X);
        x->foo(arg);   // call foo with the copy
}
```

somDefaultInit

This method initializes instance variables and attributes in a newly created object. Replaces **somInit** as the preferred method for default object initialization. For performance reasons, it is recommended that **somDefaultInit** always be overridden by classes.

Syntax

```
void somDefaultInit   (SOMObject  receiver, somInitCtrl  ctrl)
```

Parameters

receiver (SOMObject)
A pointer to an object of type **SOMObject**.

ctrl (somInitCtrl)
A pointer to a **somInitCtrl** data structure. SOMobjects uses this data structure to control the initialization of the ancestor classes, thereby ensuring that no ancestor class receives multiple initialization calls. A pointer to a class that is an ancestor of the actual class of the receiver.

Returns

rc (void)

Remarks

Every SOM class is expected to support a set of initializer methods. This set will always include **somDefaultInit**, whether or not the class explicitly overrides **somDefaultInit**. All other initializer methods for a class must be explicitly introduced by the class.

The purpose of an initializer method supported by a class is first to invoke initializer methods of ancestor classes (those ancestors that are the class's **directinitclasses**) and then to place the instance variables and attributes introduced by the class into some consistent state by loading them with appropriate values. The result is that, when an object is initialized, each class that contributes to its implementation will run some initializer method. The **somDefaultInit** method may or may not be among the initializers used to initialize a given object, but it is always available for this purpose.

Thus, the **somDefaultInit** method may be invoked on a newly created object to initialize its instance variables and attributes. The **somDefaultInit** method is more efficient than **somInit** (the method it replaces), and it also prevents multiple initializer calls to ancestor classes. The **somInit** method is now considered obsolete when writing new code, although **somInit** is still supported.

To override **somDefaultInit**, the **implementation** section of the class's .idl file should include **somDefaultInit** with the **override** and **init** modifiers specified. (The **init** modifier signifies that the method is an initializer method.) No additional coding is required for the resulting **somDefaultInit** stub procedure in the implementation template file, unless the class implementor wishes to customize object initialization in some way.

If the .idl file does not explicitly override **somDefaultInit**, then by default a generic method procedure for **somDefaultInit** will be provided by the SOMobjects Toolkit. If invoked, this generic method procedure first invokes **somDefaultInit** on the appropriate ancestor classes, and then (for consistency with earlier versions of SOMobjects) calls any **somInit** code that may have been provided by the class (if **somInit** was overridden). Because the generic procedure for **somDefaultInit** is less efficient than the stub procedure that is provided when **somDefaultInit** is overridden, it is recommended that the .idl file always override **somDefaultInit**.

Note: It is not appropriate to override both **somDefaultInit** and **somInit**. If this is done, the **somInit** code will not be executed. The best way to convert an old class that overrides **somInit** to use of the more efficient **somDefaultInit** (if this is desired) is as follows: (1) Replace the **somInit** override in the class's .idl file with an override for **somDefaultInit**, (2) run the implementation template emitter to produce a stub procedure for **somDefaultInit**, and then (3) simply call the class's **somInit** procedure directly (not using a method invocation) from the **somDefaultInit** method procedure.

As mentioned above, the object#initialization framework supported by SOMobjects allows a class to support additional initializer methods besides **somDefaultInit**. These additional initializers will typically include special#purpose arguments, so that objects of the class can be initialized with special capabilities or characteristics. For each new initializer method, the **implementation** section must include the method name with the **init** modifier. Also, the **directinitclasses** modifier can be used if, for some reason, the class implementor wants to control the order in which ancestor initializers are executed.

Note: It is recommended that the method name for an initializer method include the class name as a prefix. A newly defined initializer method will include an implicit Environment argument if the class does not use a **callstyle=oidl** modifier. There may be constraints associated with modification of the procedure stubs for initializers.

somDefaultInit

Original Class

SOMObject

Related Methods

Methods

- **somDestruct**

Example Code

```
// SOM IDL
#include <Animal.idl>

interface Dog : Animal
{
    implementation {
        releaseorder: ;
            somDefaultInit: override, init;
        };
};

(null)
Original Class
```

somDestruct

This method uninitializes the receiving object, and (if so directed) frees object storage after uninitialization has been completed. Replaces **somUninit** as the preferred method for uninitializing objects. For performance reasons, it is recommended that **somDestruct** always be overridden. Not normally invoked directly by object clients.

Syntax

```
void somDestruct  (SOMObject  receiver, octet  dofree,
                   somDestructCtrl  ctrl)
```

Parameters

receiver (SOMObject)
A pointer to an object.

dofree (octet)
A boolean indicating whether the caller wants the object storage freed after uninitialization of the current class has been completed. Passing 1 (true) indicates the object storage should be freed.

ctrl (somDestructCtrl)
A pointer to a **somDestructCtrl** data structure. SOMobjects uses this data structure to control the uninitialization of the ancestor classes, thereby ensuring that no ancestor class receives multiple uninitialization calls. If a user invokes **somDestruct** on an object directly, a NULL (that is, zero) ctrl pointer can be passed. This instructs the receiving code to obtain a **somDestructCtrl** data structure from the class of the object.

Returns

rc (void)

Remarks

Every class must support the **somDestruct** method. This is accomplished either by overriding **somDestruct** (in which case a specialized stub procedure will be generated in the implementation template file), or else **SOMobjects** will automatically provide a generic procedure that implements **somDestruct** for the class. The generic procedure

calls **somUninit** (if this was overridden) to perform local uninitialization, then completes execution of the method appropriately.

Because the specialized stub procedure generated by the template emitter is more efficient than the generic procedure provided when **somDestruct** is not overridden, it is recommended that **somDestruct** always be overridden. The stub procedure that is generated in this case requires no modification for correct operation. The only modification appropriate within this stub procedure is to uninitialize locally introduced instance variables. See Section 5.5, 'Initializing and Unitializing Objects,' of the *SOM Programming Guide* for further details.

Uninitialization with **somDestruct** executes as follows: For any given class in the ancestor chain, **somDestruct** first uninitializes that class's introduced instance variables (if this is appropriate), and then calls the next ancestor class's implementation of **somDestruct**, passing 0 (that is, false) as the interim dofree argument. Then, after all ancestors of the given class have been uninitialized, if the class's own **somDestruct** method were originally invoked with dofree as 1 (that is, true), then that object's storage is released.

Note: It is not appropriate to override both **somDestruct** and **somUninit**. If this is done, the **somUninit** code will not be executed. The best way to convert an old class that overrides **somUninit** to use of the more efficient **somDestruct** (if this is desired) is as follows: (1) Replace the **somUninit** override in the class's .idl file with an override for **somDestruct**, (2) run the emitter to produce a stub procedure for **somDestruct** in the implementation template file, and then (3) simply call the class's **somUninit** procedure directly (not using a method invocation) from the **somDestruct** procedure.

Original Class

SOMObject

Related Methods

Functions

- **somDefaultInit**

Example Code

```
// SOM IDL
#include <Animal.idl>

interface Dog : Animal
{
    implementation {
        releaseorder: ;
            somDestruct: override;
        };
};
```

somDispatch

Invokes a method using dispatch method resolution. The **somDispatch** method is designed to be overridden. The **somClassDispatch** method is not generally overridden.

For backward compatibility, this method does *not* take an **Environment** parameter.

Syntax

```
boolean somDispatch  (SOMObject  receiver, somToken  retValue,
                      somID  methodId, va_list  args)
```

Parameters

receiver (SOMObject)
A pointer to the object whose class will be used for method resolution by **somDispatch**.

retValue (somToken)
The address of the area in memory where the result of the invoked method procedure is to be stored. The caller is responsible for allocating enough memory to hold the result of the specified method. When dispatching methods that return no result (i.e., void), a NULL may be passed as this argument.

methodId (somID)
A **somId** identifying the method to be invoked. A string representing the method name can be converted to a **somId** using the **somIdFromString** function.

args (va_list)
A **va_list** containing the arguments to be passed to the method identified by *methodId*. The arguments must include a pointer to the target object as the first entry. As a convenience for C and C++ programmers, SOM's language bindings provide a varargs invocation macro for va_list methods (such as **somDispatch** and **somClassDispatch).** The example below illustrates this.

Returns

rc (boolean)
A boolean representing whether or not the method was successfully dispatched is returned. The reason for this is that **somDispatch** and **somClassDispatch** use the function **somApply** to invoke the resolved method procedure, and **somApply**

requires an apply stub for successful execution. In support of old class binaries SOM does not consider a NULL apply stub to be an error. As a result, somApply may fail. If this happens, then false is returned; otherwise true is returned.

Remarks

somDispatch and **somClassDispatch** perform method resolution to select a method procedure, and then invoke this procedure on *args*. The somSelf argument for the selected method procedure (called the *target object*, below, to distinguish it from the receiver of the **somDispatch** or **somClassDispatch** method call) is the first argument included in the va_list, *args*.

For **somDispatch**, method resolution is performed using the class of the receiver; for **somClassDispatch**, method resolution is performed using the argument class, *clsObj*. Because **somClassDispatch** uses *clsObj* for method resolution, a programmer invoking **somDispatch** or **somClassDispatch** should assure that the class of the target object is either derived from or is identical to the class used for method resolution; otherwise, a runtime error will likely result when the target object is passed to the resolved procedure. Although not necessary, the receiver is usually also the target object.

The **somDispatch** and **somClassDispatch** methods supersede the **somDispatch***X* methods. Unlike the **somDispatch***X* methods, which are restricted to few return types, the **somDispatch** and **somClassDispatch** methods make no assumptions concerning the result returned by the method to be invoked. Thus, **somDispatch** and **somClassDispatch** can be used to invoke methods that return structures. The **somDispatch***X* methods now invoke **somDispatch**, so overriding **somDispatch** serves to override the **somDispatch***X* methods as well.

Original Class

SOMObject

Related Methods

Functions

- **somApply**

Example Code

Given class *Key* that has an attribute *keyval* of type **long** and an overridden method for **somPrintSelf** that prints the value of the attribute (as well as the information printed by **SOMObject**'s implementation of **somPrintSelf**), the following client code invokes methods on *Key* objects using **somDispatch** and **somClassDispatch**. (The *Key* class was defined with the **callstyle=oidl** class modifier, so the **Environment** argument is not required of its methods.)

somDispatch

```
#include <key.h>

main()
{
  SOMObject obj;
  long k1 = 7, k2;
  Key *myKey = KeyNew();
  somVaBuf vb;
  va_list push, args;

  somId setId = somIdFromString("_set_keyval");
  somId getId = somIdFromString("_get_keyval");
  somId prtId = somIdFromString("somPrintSelf");

  vb = (somVaBuf)somVaBuf_create(NULL, 0);
  somVaBuf_add(vb, (char *)&myKey, tk_ulong);
  somVaBuf_add(vb, (char *)&k1, tk_long);
  somVaBuf_get_valist(vb, &args);


  /* va_list invocation of setkey and getkey */
  SOMObject_somDispatch(myKey, (somToken *)0, setId, args);
  somVaBuf_get_valist(vb, &args);
  SOMObject_somDispatch(myKey, (somToken *)&k2, getId, args);
  printf("va_list _set_keyval and _get_keyval: %i\n", k2);

  /* varargs invocation of setkey and getkey */
  _somDispatch(myKey, (somToken *)0, setId, myKey, k1);
  _somDispatch(myKey, (somToken *)&k2, getId, myKey);
  printf("varargs _set_keyval and _get_keyval: %i\n", k2);

  /* illustrate somClassDispatch "casting" (use varargs form) */
  printf("somPrintSelf on myKey as a Key:\n");
  _somClassDispatch(myKey, _Key, (somToken *)&obj, prtId, myKey, 0);

  printf("somPrintSelf on myKey as a SOMObject:\n");
  _somClassDispatch(myKey, _SOMObject, (somToken *)&obj, prtId, myKey, 0);

  SOMFree(setId);
  SOMFree(getId);
  SOMFree(prtId);
  _somFree(myKey);

  somVaBuf_destroy(vb);
}
```

This program produces the following output:

```
va_list _set_keyval and _get_keyval: 7
varargs _set_keyval and _get_keyval: 7
somPrintSelf on myKey as a Key:
{An instance of class Key at address 2005B2F8}
    -- with key value 7
somPrintSelf on myKey as a SOMObject:
{An instance of class Key at address 2005B2F8}
```

somClassDispatch

Invokes a method using dispatch method resolution. The **somDispatch** method is designed to be overridden. The **somClassDispatch** method is not generally overridden.

For backward compatibility, this method does *not* take an **Environment** parameter.

Syntax

```
boolean somClassDispatch  (SOMObject  receiver, SOMClass  clsObj,
                           somToken  retValue, somID  methodId,
                           va_list  args)
```

Parameters

receiver (SOMObject)
A pointer to the object whose class will be used for method resolution by **somDispatch**.

clsObj (SOMClass)
A pointer to the class that will be used for method resolution by **somClassDispatch**.

retValue (somToken)
The address of the area in memory where the result of the invoked method procedure is to be stored. The caller is responsible for allocating enough memory to hold the result of the specified method. When dispatching methods that return no result (i.e., void), a NULL may be passed as this argument.

methodId (somID)
A **somId** identifying the method to be invoked. A string representing the method name can be converted to a **somId** using the **somIdFromString** function.

args (va_list)
A **va_list** containing the arguments to be passed to the method identified by *methodId*. The arguments must include a pointer to the target object as the first entry. As a convenience for C and C++ programmers, SOM's language bindings provide a varargs invocation macro for va_list methods (such as **somDispatch** and **somClassDispatch).** The example below illustrates this.

Returns

rc (boolean)

A boolean representing whether or not the method was successfully dispatched is returned. The reason for this is that **somDispatch** and **somClassDispatch** use the function **somApply** to invoke the resolved method procedure, and **somApply** requires an apply stub for successful execution. In support of old class binaries SOM does not consider a NULL apply stub to be an error. As a result, somApply may fail. If this happens, then false is returned; otherwise true is returned.

Remarks

somDispatch and **somClassDispatch** perform method resolution to select a method procedure, and then invoke this procedure on *args*. The somSelf argument for the selected method procedure (called the *target object*, below, to distinguish it from the receiver of the **somDispatch** or **somClassDispatch** method call) is the first argument included in the va_list, *args*.

For **somDispatch**, method resolution is performed using the class of the receiver; for **somClassDispatch**, method resolution is performed using the argument class, *clsObj*. Because **somClassDispatch** uses *clsObj* for method resolution, a programmer invoking **somDispatch** or **somClassDispatch** should assure that the class of the target object is either derived from or is identical to the class used for method resolution; otherwise, a runtime error will likely result when the target object is passed to the resolved procedure. Although not necessary, the receiver is usually also the target object.

The **somDispatch** and **somClassDispatch** methods supersede the **somDispatch***X* methods. Unlike the **somDispatch***X* methods, which are restricted to few return types, the **somDispatch** and **somClassDispatch** methods make no assumptions concerning the result returned by the method to be invoked. Thus, **somDispatch** and **somClassDispatch** can be used to invoke methods that return structures. The **somDispatch***X* methods now invoke **somDispatch**, so verriding **somDispatch** serves to override the **somDispatch***X* methods as well.

Original Class

SOMObject

Related Methods

Functions

- **somApply**

Example Code

Given class *Key* that has an attribute *keyval* of type **long** and an overridden method for **somPrintSelf** that prints the value of the attribute (as well as the information printed by **SOMObject**'s implementation of **somPrintSelf**), the following client code invokes methods on *Key* objects using **somDispatch** and **somClassDispatch**. (The *Key* class was defined with the **callstyle=oidl** class modifier, so the **Environment** argument is not required of its methods.)

```
#include <key.h>

main()
{
  SOMObject obj;
  long k1 = 7, k2;
  Key *myKey = KeyNew();
  somVaBuf vb;
  va_list push, args;

  somId setId = somIdFromString("_set_keyval");
  somId getId = somIdFromString("_get_keyval");
  somId prtId = somIdFromString("somPrintSelf");

  vb = (somVaBuf)somVaBuf_create(NULL, 0);
  somVaBuf_add(vb, (char *)&myKey, tk_ulong);
  somVaBuf_add(vb, (char *)&k1, tk_long);
  somVaBuf_get_valist(vb, &args);


  /* va_list invocation of setkey and getkey */
  SOMObject_somDispatch(myKey, (somToken *)0, setId, args);
  somVaBuf_get_valist(vb, &args);
  SOMObject_somDispatch(myKey, (somToken *)&k2, getId, args);
  printf("va_list _set_keyval and _get_keyval: %i\n", k2);

  /* varargs invocation of setkey and getkey */
  _somDispatch(myKey, (somToken *)0, setId, myKey, k1);
  _somDispatch(myKey, (somToken *)&k2, getId, myKey);
  printf("varargs _set_keyval and _get_keyval: %i\n", k2);

  /* illustrate somClassDispatch "casting" (use varargs form) */
  printf("somPrintSelf on myKey as a Key:\n");
  _somClassDispatch(myKey, _Key, (somToken *)&obj, prtId, myKey, 0);

  printf("somPrintSelf on myKey as a SOMObject:\n");
  _somClassDispatch(myKey, _SOMObject, (somToken *)&obj, prtId, myKe

  SOMFree(setId);
  SOMFree(getId);
  SOMFree(prtId);
  _somFree(myKey);

  somVaBuf_destroy(vb);
}
```

This program produces the following output:

```
va_list _set_keyval and _get_keyval: 7
varargs _set_keyval and _get_keyval: 7
somPrintSelf on myKey as a Key:
{An instance of class Key at address 2005B2F8}
    -- with key value 7
somPrintSelf on myKey as a SOMObject:
{An instance of class Key at address 2005B2F8}
```

somDumpSelf

This method writes out a detailed description of the receiving object. Intended for use by object clients. Not generally overridden. Note: For backward compatibility, this method does *not* take and Environment parameter.

Syntax

```
void somDumpSelf  (SOMObject  receiver, long  level)
```

Parameters

receiver (SOMObject)
A pointer to the object to be dumped.

level (long)
The nesting level for describing compound objects. It must be greater than or equal to 0. All lines in the description will be preceded by "2 * level" spaces.

Returns

rc (void)

Remarks

The **somDumpSelf** method performs some initial setup, and then invokes the **somDumpSelfInt** method to write a detailed description of the receiver, including its state.

Original Class

SOMOBject

Related Methods

Methods

- **somDumpSelfInt**

somDumpSelfInt

This method outputs the internal state of an object. Intended to be overridden by class implementors. Not intended to be directly invoked by object clients.

For backward compatibility, this method does *not* take an **Environment** parameter.

Syntax

```
void somDumpSelfInt  (SOMObject  receiver, long  level)
```

Parameters

receiver (SOMObject)
A pointer to the object to be dumped.

level (long)
The nesting level for describing compound objects. It must be greater than or equal to 0. All lines in the description should be preceded by "2* level" spaces.

Returns

rc (void)

Remarks

The **somDumpSelfInt** method should be overridden by a class implementor, to write out the instance data stored in an object. This method is invoked by the **somDumpSelf** method, which is used by object clients to output the state of an object.

The procedure used to override this method for a new class should begin by calling the parent class form of this method on each of the class parents, and should then write a description of the instance variables introduced by new class. This will result in a description of all the class's instance variables. The C and C++ implementation bindings provide a convenient macro for performing parent method calls on all parents, as illustrated below.

The character output routine pointed to by **SOMOutCharRoutine** should be used for output. The **somLPrintf** function is especially convenient for this, since level is handled appropriately.

Original Class

SOMObject

Related Methods

Methods

- **somDumpSelf**
- **somPrintSelf**

Example Code

Below is a method overriding **somDumpSelfInt** for class "List", which has two attributes, *val* (which is a **long**) and *next* (which is a pointer to a "List" object).

```
SOM_Scope void   SOMLINK somDumpSelfInt(List somSelf, int level)
{
    ListData *somThis = ListGetData(somSelf);
    Environment *ev = somGetGlobalEnvironment();

    List_parents_somDumpSelfInt(somSelf, level);
    somLPrintf(level, "This item: %i\n", __get_val(somSelf, ev);
    somLPrintf(level, "Next item: \n");
    if (__get_next(somSelf, ev) != (List) NULL)
        _somDumpSelfInt(__get_next(somSelf, ev), level+1);
    else
        somLPrintf(level+1, "NULL\n");
}
```

somDumpSelfInt

Below is a client program that invokes the **somDumpSelf** method on "List" objects.

```
#include <list.h>

main()
{
   List L1, L2;
   long x = 7, y = 13;
   Environment *ev = somGetGlobalEnvironment();

   L1 = ListNew();
   L2 = ListNew();
   __set_val(L1, ev, x);
   __set_next(L1, ev, (List) NULL);
   __set_val(L2, ev, y);
   __set_next(L2, ev, L1);

   _somDumpSelf(L2,0);

   _somFree(L1);
   _somFree(L2);
}
```

Below is the output produced by this program:

```
{An instance of class List at 0x2005EA8
 This item: 13
 Next item:
   1 This item: 7
   1 Next item:
     2 NULL
}
```

somFree

This method releases the storage used by an object. Intended for use by object clients. Not generally overridden.

For backward compatibility, this method does *not* take an **Environment** parameter.

Syntax

```
void somFree   (SOMObject  receiver)
```

Parameters

receiver (SOMObject)
A pointer to the object to be freed.

Returns

rc (void)

Remarks

The **somFree** method releases the storage containing the receiver object by calling the method **somDeallocate**. No future references should be made to the receiver once this is done. The **somFree** method calls **somDestruct** to allow storage pointed to by the object to be freed.

The **somFree** method should not be called on objects created by **somRenew**, thus the method is normally only used by code that also created the object.

Note: SOM also supplies a macro, **SOMFree**, which is used to free a block of memory. This macro should not be used on objects.

Original Class

SOMObject

Related Methods

Methods

- **somNew**
- **somNewNoInit**

- **somDestruct**

Functions

- SOMFree

Example Code

```
#include <animal.h>

void main()
{
   Animal myAnimal;
   /*
    * Create an object.
    */
   myAnimal = AnimalNew();

   /* ... */

   /* Free it when finished. */
   _somFree(myAnimal);
}
```

somGetClass

This method returns a pointer to an object's class object. Not generally overridden.

For backward compatibility, this method does *not* take an **Environment** parameter.

Syntax

```
SOMClass somGetClass   (SOMObject  receiver)
```

Parameters

receiver (SOMObject)
A pointer to the object whose class is desired.

Returns

rc (SOMClass)
A pointer to the object's class object. This return value is cast as a **(SOMClass*)**. In C++, you may have to explicitly cast this to a pointer of a specific class type (when different from **SOMClass**).

Remarks

somGetClass obtains a pointer to the receiver's class object. The **somGetClass** method is typically not overridden.

Important Note: For C and C++ programmers, SOM provides a **SOM_GetClass** macro that performs the same function. This macro should only be used **only** when absolutely necessary (i.e., when a method call on the object is not possible), since it bypasses whatever semantics may be intended for the somGetClass method by the implementor of the receiver's class. Even class implementors do not know whether a special semantics for this method is inherited from ancestor classes. If you are unsure of whether the method or the macro is appropriate, you should use the method call.

Original Class

SOMObject

somGetClass

Related Methods

Methods

- **SOM_GetClass**

Example Code

```
#include <animal.h>
main()
{
  Animal myAnimal;
  int numMethods;
  SOMClass animalClass;

  myAnimal = AnimalNew ();
  animalClass = _somGetClass (myAnimal);
  SOM_Test(animalClass == _Animal);
}
```

somGetClassName

This method returns the name of the class of an object. Not generally overridden.

For backward compatibility, this method does *not* take an **Environment** parameter.

Syntax

```
string somGetClassName   (somObject  receiver)
```

Parameters

receiver (somObject)
A pointer to the object whose class name is desired.

Returns

rc (string)
Returns a pointer to the name of the class.

Remarks

The **somGetClassName** method returns a pointer to a zero-terminated string that gives the name of the class of an object.

This method is not generally overridden; it simply invokes **somGetName** on the class of the receiver. Refer to **somGetName** for more information on the returned string.

Original Class

SOMObject

Related Methods

Methods

- **somGetName**

Example Code

```
#include <animal.h>
main()
{
  Animal myAnimal;
  SOMClass animalClass;
  char *className;

  myAnimal = AnimalNew();
  className = _somGetClassName(myAnimal);
  somPrintf("Class name: %s\n", className);
  _somFree(myAnimal);
}
/*
Output from this program:
Class name:  Animal
*/
```

somGetSize

This method returns the size of an object. Not generally overridden.

For backward compatibility, this method does *not* take an **Environment** parameter.

Syntax

```
long somGetSize   (SOMObject  receiver)
```

Parameters

receiver (SOMObject)
A pointer to the object whose size is desired.

Returns

rc (long)
Returns the size, in bytes, of the receiver.

Remarks

The **somGetSize** method returns the total amount of contiguous space used by the receiving object.

The value returned reflects only the amount of storage needed to hold the SOM representation of the object. The object might actually be using or managing additional space outside of this area.

The **somGetSize** method is not generally overridden.

Original Class

SOMObject

Related Methods

Methods

- **somGetInstancePartSize**
- **somGetInstanceSize**

Example Code

```
#include <animal.h>
void main()
{
   Animal myAnimal;
   long animalSize;
   myAnimal = AnimalNew();
   animalSize = _somGetSize(myAnimal);
   somPrintf("Size of animal (in bytes): %d\n", animalSize);
   _somFree(myAnimal);
}
/*
Output from this program:
Size of animal (in bytes): 8
*/
```

somInit

This method initializes instance variables or attributes in a newly created object. Designed to be overridden. Note: The newer **somDefaultInit** method is suggested instead.

For backward compatibility, this method does *not* take an **Environment** parameter.

Syntax

```
void somInit   (SOMObject  receiver)
```

Parameters

receiver (SOMObject)
A pointer to the object to be initialized.

Returns

rc (void)

Remarks

The **somInit** method is invoked to cause a newly created object to initialize its instance variables or attributes.

Note: The newer **somDefaultInit** method performs object initialization more efficiently and is now the preferred approach for overriding initialization in an implementation file. (The **somInit** method still executes correctly as before.)

Because instances of **SOMObject** do not have any instance data, the default implementation does nothing. It is provided as a convenience to class implementors so that initialization of objects can be done in a uniform way across all classes (by overriding **somInit**). This method is called automatically by **somNew** during object creation.

A companion method, **somUninit**, is called whenever an object is freed. These two methods should be designed to work together, with **somInit** priming an object for its first use, and **somUninit** preparing the object for subsequent release.

somInit

If objects of your class contain instance variables or attributes, override the **somInit** method to initialize the instance variables or attributes when instances of the class are created. When overriding this method, always call all parent (base) classes' versions of this method *before* doing your own initialization, as follows:

1. The overriding implementation should invoke the parent method for *each* parent. For users of the C or C++ implementation bindings, this can be done in either of two ways: (a) by calling a **_parents_** macro (which automatically invokes all parent methods) or (b) by calling the **_parent_** *parentName*>**_** macro on each parent separately.

For more information on parent method calls, see the topic "Extending the Implementation Template" in Chapter 5, "Implementing Classes in SOM," of the *SOM Programming Guide*.

2. The code must be written so that it can be executed multiple times without harm on the same object. This is necessary because, under multiple inheritance, parent method calls that progress up the inheritance hierarchy may encounter the same ancestor class more than once (where different inheritance paths "join" when followed backward). A check can be made to determine whether a particular invocation of **somInit** is the first on a given object by examining the contents of its instance variables; all the instance variables and attributes of a newly created SOM object are set to zero before **somInit** is invoked on that object.

More information and examples on object initialization (especially regarding the **somDefaultInit** method) are given in the topic "Initializing and Uninitializing Objects" in Chapter 5, "Implementing Classes in SOM," of the *SOM Programming Guide*.

Original Class

SOMObject

Related Methods

Methods

- somDefaultInit
- **somNew**
- **somRenew**
- **somDestruct**
- **somUninit**

Example Code

Below is the implementation for a class *Animal* that introduces an attribute *sound* of type *string* and overrides **somInit** and **somUninit,** along with a main program that creates and then frees an instance of class *Animal*.

```
#define Animal_Class_Source
 #include <animal.ih>
 #include <string.h>

 SOM_Scope void SOMLINK somInit (Animal somSelf)
 {
      AnimalData *somThis = AnimalGetData (somSelf);
      Environment *ev = somGetGlobalEnvironment();
      Animal_parents_somInit (somSelf);
      if (!__get_sound(somSelf, ev)) {
         __set_sound(somSelf, ev, SOMMalloc(100));
         strcpy (__get_sound(somSelf, ev), "Unknown Noise");
         somPrintf ("New Animal Initialized\n");
      }
 }

 SOM_Scope void SOMLINK somUninit (Animal somSelf)
  {
      AnimalData *somThis = AnimalGetData (somSelf);
      Environment *ev = somGetGlobalEnvironment();
      if (__get_sound(somSelf, ev)) {
        SOMFree(__get_sound(somSelf, ev);
        __set_sound(somSelf, ev, (char*)0);
        somPrintf ("Animal Uninitialized\n");
        Animal_parents_somUninit (somSelf);
      }
}

/* main program */
   #include <animal.h>
   void main()
   {
      Animal myAnimal;
      myAnimal = AnimalNew ();
      _somFree (myAnimal);
   }

/*
Program output:
New Animal Initialized
Animal Uninitialized
*/
```

somIsA

This method tests whether an object is an instance of a given class or of one of its descendant classes. Not generally overridden.

For backward compatibility, this method does *not* take an **Environment** parameter.

Syntax

```
boolean somIsA   (SOMObject  receiver, SOMClass  aClass)
```

Parameters

receiver (SOMObject)
A pointer to the object to be tested.

aClass (SOMClass)
A pointer to the class that the object should be tested against.

Returns

rc (boolean)
Returns 1 (true) if the receiving object is an instance of the specified class or (unlike **somIsInstanceOf**) of any of its descendant classes, and 0 (false) otherwise.

Remarks

Use the **somIsA** method to determine if an object can be treated like an instance of *aClass*. SOM guarantees that if somIsA returns true, then the receiver will respond to all (static or dynamic) methods supported by *aClass*.

Original Class

SOMObject

Related Methods

Methods

- **somIsDescendedFrom**
- **somIsInstanceOf**
- **somRespondsTo**
- **somSupportsMethod**

Example Code

```
#include <dog.h>
/* ---------------------------------
   Note:  Dog is derived from Animal.
   --------------------------------- */
main()
{
  Animal myAnimal;
  Dog myDog;
  SOMClass animalClass;
  SOMClass dogClass;

  myAnimal = AnimalNew();
  myDog = DogNew();
  animalClass = _somGetClass (myAnimal);
  dogClass = _somGetClass (myDog);
  if (_somIsA (myDog, animalClass))
     somPrintf ("myDog IS an Animal\n");
  else
     somPrintf ("myDog IS NOT an Animal\n");
  if (_somIsA (myAnimal, dogClass))
     somPrintf ("myAnimal IS a Dog\n");
  else
     somPrintf ("myAnimal IS NOT a Dog\n");
  _somFree (myAnimal);
  _somFree (myDog);
}
/*
Output from this program:
myDog IS an Animal
myAnimal IS NOT a Dog
*/
```

somIsInstanceOf

This method determines whether an object is an instance of a specific class. Not generally overridden.

For backward compatibility, this method does *not* take an **Environment** parameter.

Syntax

```
boolean somIsInstanceOf  (SOMObject  receiver, SOMClass  aClass)
```

Parameters

receiver (SOMObject)
A pointer to the object to be tested.

aClass (SOMClass)
A pointer to the class that the object should be an instance of.

Returns

rc (boolean)
Returns 1 (true) if the receiving object is an instance of the specified class, and 0 (false) otherwise.

Remarks

Use the **somIsInstanceOf** method to determine if an object is an instance of a specific class. This method tests an object for inclusion in one specific class. It is equivalent to the expression:

```
(aClass == somGetClass (receiver))
```

Original Class

SOMObject

Related Methods

Methods

- **somDescendedFrom**
- **somIsA**

Example Code

```
#include <dog.h>
/* ---------------------------------
   Note:  Dog is derived from Animal.
   --------------------------------- */
main()
{
  Animal myAnimal;
  Dog myDog;
  SOMClass animalClass;
  SOMClass dogClass;

  myAnimal = AnimalNew ();
  myDog = DogNew ();
  animalClass = _somGetClass (myAnimal);
  dogClass = _somGetClass (myDog);
  if (_somIsInstanceOf (myDog, animalClass))
     somPrintf ("myDog is an instance of Animal\n");
  if (_somIsInstanceOf (myDog, dogClass))
     somPrintf ("myDog is an instance of Dog\n");
  if (_somIsInstanceOf (myAnimal, animalClass))
     somPrintf ("myAnimal is an instance of Animal\n");
  if (_somIsInstanceOf (myAnimal, dogClass))
     somPrintf ("myAnimal is an instance of Dog\n");
  _somFree (myAnimal);
  _somFree (myDog);
}
/*
Output from this program:
myDog is an instance of Dog
myAnimal is an instance of Animal
*/
```

somPrintSelf

This method outputs a brief description that identifies the receiving object. Designed to be overridden.

Note: For backward compatibility, this method does *not* take an **Environment** parameter.

Syntax

```
SOMObject somPrintSelf  (SOMObject  receiver)
```

Parameters

receiver (SOMObject)
A pointer to the object to be described.

Returns

rc (SOMObject)
Returns a pointer to the receiver object as its result.

Remarks

somPrintSelf should output a brief string containing key information useful to identify the receiver object, rather than a complete dump of the receiver object state as provided by **somDumpSelfInt**. The **somPrintSelf** method should use the character output routine **SOMOutCharRoutine** (or any of the **somPrintf** functions) for this purpose. The default implementation outputs the name of the receiver object's class and the receiver's address in memory.

Because the most specific identifying information for an object will often be found within instance data introduced by the class of an object, it is likely that a class implementor that overrides this method will not need to invoke parent methods in order to provide a useful string identifying the receiver object.

Original Class

SOMObject

Related Methods

Methods

- **somDumpSelf**
- **somDumpSelfInt**

Example Code

```
#include <animal.h>
main()
{
  Animal myAnimal;
  myAnimal = AnimalNew ();
  /* ... */
  _somPrintSelf (myAnimal);
  _somFree (myAnimal);
}
/*
Output from this program:

{An instance of class Animal at address 0001CEC0}
*/
```

somResetObj

Resets an object's class to its true class after use of the **somCastObj** method.

Syntax

```
boolean somResetObj   (SOMObject  receiver)
```

Parameters

receiver (SOMObject)
A pointer to a SOM object.

Returns

rc (boolean)
The **somResetObj** method returns 1 (TRUE) always.

Remarks

The **somResetObj** method resets an object's class to its true class after use of the **somCastObj** method.

Original Class

SOMObject

Related Methods

Methods

- **somCastObj**

Example Code

```
#include <som.h>
main()
{
   SOMClassMgr cm = somEnvironmentNew();
   SOM_Test(1 == _somCastObj(cm, _SOMObject));
   _somDumpSelf(cm, 0));
   SOM_Test(1 == _somResetObj(cm));
   _somDumpSelf(cm, 0);
}

(null)
/* output:
 *  {An instance of class SOMClassMgr->SOMObject
 *   at address 20061268
 *  }
 *  {An instance of class SOMClassMgr at address 20061268
 *   ... <SOMClassMgr State Information> ...
 *  }
 */
```

somRespondsTo

This method tests whether the receiving object supports a given method. Not generally overridden. Note: For backward compatibility, this method does *not* take an Environment parameter.

Syntax

```
boolean somRespondsTo  (SOMObject  receiver, somId  methodId)
```

Parameters

receiver (SOMObject)
A pointer to the object to be tested.

methodId (somId)
A **somId** that represents the name of the desired method.

Returns

rc (boolean)
Returns TRUE if the specified method can be invoked on the receiving object, and FALSE otherwise.

Remarks

The **somRespondsTo** method tests whether a specific (static or dynamic) method can be invoked on the receiver object. This test is equivalent to determining whether the class of the receiver *supports* the specified method on its instances.

Original Class

SOMObject

Related Methods

Methods

- **somSupportsMethod**

Example Code

```
/* ----------------------------------------
   Note:  Animal supports a setSound method;
   Animal does not support a doTrick method.
   ---------------------------------------- */
#include <animal.h>
main()
{
  Animal myAnimal;
  char *methodName1 = "setSound";
  char *methodName2 = "doTrick";

  myAnimal = AnimalNew();
  if (_somRespondsTo(myAnimal, SOM_IdFromString(methodName1)))
     somPrintf("myAnimal responds to %s\n", methodName1);
  if (_somRespondsTo(myAnimal, SOM_IdFromString(methodName2)))
     somPrintf("myAnimal responds to %s\n", methodName2);
  _somFree(myAnimal);
}
/*
Output from this program:
myAnimal responds to setSound
*/
```

somUninit

This method un-initializes the receiving object. Designed to be overridden by class implementors. Not normally invoked directly by object clients.

Note: For backward compatibility, this method does *not* take an **Environment** parameter.

Syntax

```
void somUninit   (SOMObject  receiver)
```

Parameters

receiver (SOMObject)
A pointer to the object to be un-initialized.

Returns

rc (void)

Remarks

The **somUninit** method performs the inverse of object initialization. Class implementors that introduce instance data that points to allocated storage should override somUninit so allocated storage can be freed when an object is freed.

This method is called automatically by **somFree** to clean up anything necessary (such as extra storage dynamically allocated to the object) before **somFree** releases the storage allocated to the object itself.

Code responsible for freeing an object must first know that there will be no further references to this object. Once this is known, this code would normally invoke **somFree** (which calls **somUninit**). In cases where **somRenew** was used to create an object instance, however, **somFree** cannot be called (e.g., the storage containing the object may simply be a location on the stack), and in this case, **somUninit** must be called explicitly.

When overriding this method, always call the parent-class versions of this method *after* doing your own un-initialization. Furthermore, just as with **somInit**, because your method may be called multiple times (due to multiple inheritance), you should

zero out references to memory that is freed, and check for zeros before freeing memory and calling the parent methods.

Original Class

SOMObject

Related Methods

Methods

- **somInit**
- **somNew**
- **somRenew**

Example Code

Following is the implementation for a class *Animal* that introduces an attribute *sound* of type *string* and overrides **somInit** and **somUninit,** along with a main program that creates and then frees an instance of class *Animal*.

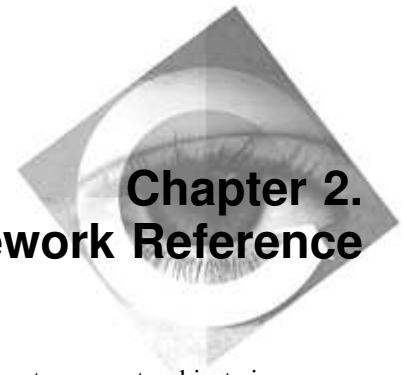
somUninit

```
#define Animal_Class_Source
#include <animal.ih>
#include <string.h>

SOM_Scope void SOMLINK somInit (Animal somSelf)
 {
     AnimalData *somThis = AnimalGetData (somSelf);
     Environment *ev = somGetGlobalEnvironment();
     Animal_parents_somInit (somSelf);
     if (!__get_sound(somSelf, ev)) {
        __set_sound(somSelf, ev, SOMMalloc(100));
        strcpy (__get_sound(somSelf, ev), "Unknown Noise");
        somPrintf ("New Animal Initialized\n");
     }
 }

SOM_Scope void SOMLINK  somUninit (Animal somSelf)
 {
     AnimalData *somThis = AnimalGetData (somSelf);
     Environment *ev = somGetGlobalEnvironment();
     if (__get_sound(somSelf, ev)) {
        SOMFree(__get_sound(somSelf, ev);
        __set_sound(somSelf, ev, (char*)0);
        somPrintf ("Animal Uninitialized\n");
        Animal_parents_somUninit (somSelf);
     }
 }
/* main program */
   #include <animal.h>
   void main()
   {
      Animal myAnimal;
      myAnimal = AnimalNew ();
      _somFree (myAnimal);
   }

/*
Program output:
New Animal Initialized
Animal Uninitialized
*/
```



Chapter 2. DSOM Framework Reference

A note on DSOM and CORBA

Distributed SOM (DSOM) is a framework which supports access to objects in a distributed application. DSOM can be viewed as both:

- an extension to basic SOM facilities
- an implementation of the "Object Request Broker" (ORB) technology defined by the Object Management Group (OMG), in the Common Object Request Broker Architecture (CORBA) specification and standard, Revision 1.1. The CORBA 1.1 specification is published by x/Open and the Object Management Group (OMG).

  Note: Portions of this discussion of DSOM pertain to Workgroup DSOM only and are not supported by this Windows version of the product. References to topics such as ORB, CORBA, and selection of servers do not apply to this version. These discussions are presented here to provide a more complete understanding of DSOM in general.

One of the primary contributions of CORBA is the specification of basic runtime interfaces for writing portable, distributable object-oriented applications. SOM and DSOM implement those runtime interfaces, according to the CORBA specification.

In addition to the published CORBA 1.1 interfaces, it was necessary for to DSOM introduce several of its own interfaces, in those areas where:

- CORBA 1.1 did not specify the full interface (e.g., **ImplementationDef**, **Principal**)
- CORBA 1.1 did not address the function specified by the interface (e.g., "lifecycle" services for object creation and deletion)
- the functionality of a CORBA 1.1 interface has been enhanced by DSOM.

Any such interfaces have been noted on the reference page for each DSOM class.

A note on method naming conventions

The SOM Toolkit frameworks (including DSOM) and CORBA have slightly different conventions for naming methods. Methods introduced by the SOM Toolkit frameworks use prefixes to indicate the framework to which each method belongs, and use capitalization to separate words in the method names (for example, **somdFindServer**). Methods introduced by CORBA have no prefixes, are all lower

© Copyright IBM Corp. 1994, 1996

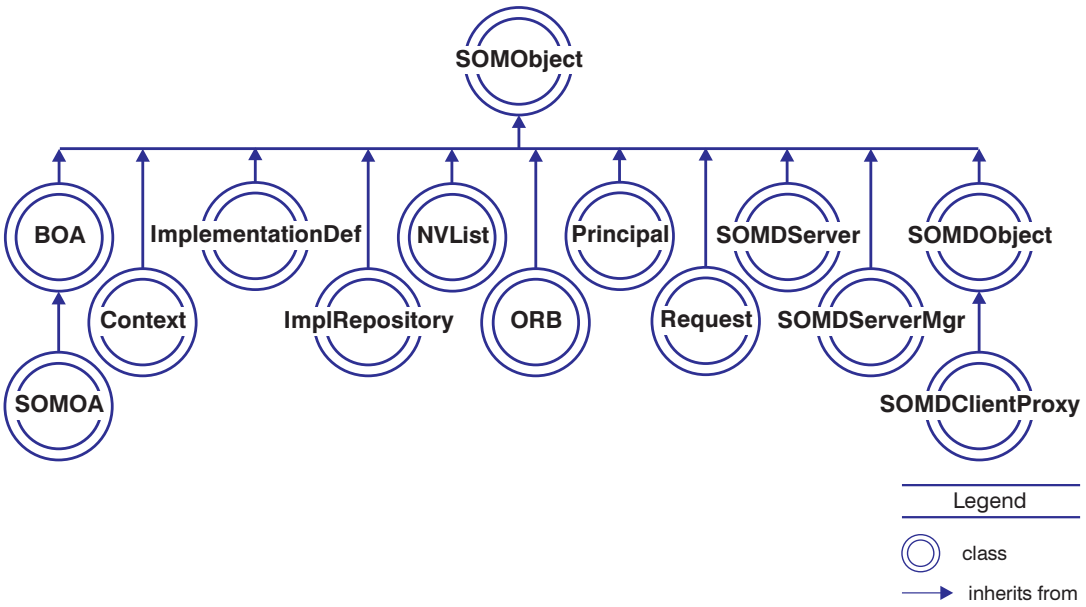


Figure 2. DSOM Framework Class Organization

case, and use underscores to separate words in the method names (such as, **impl_is_ready**).

DSOM, more than the other SOM Toolkit frameworks, uses a mix of both conventions. The method and class names introduced by CORBA 1.1 are implemented as specified, for application portability. Methods introduced by DSOM to enhance a CORBA-defined class also use the CORBA naming style. The SOM Toolkit convention for method naming is used for non-CORBA classes which are introduced by DSOM.

get_next_response

This function returns the next **Request** object to complete, after starting multiple requests in parallel.

Syntax

```
ORBStatus get_next_response (Environment  *env, Flags  response_flags,
                             Request  *req)
```

Parameters

env (Environment *)
A pointer to the **Environment** structure for the caller.

response_flags (Flags)
A **Flags** (unsigned long) variable, used to indicate whether the caller wants to wait for the next request to complete (0), or not wait (RESP_NO_WAIT).

req (Request *)
A pointer to a **Request** object variable. The address of the next **Request** object which completes is returned in the **Request** variable.

Returns

rc (ORBStatus)
May return a non-zero **ORBStatus** value, which indicates a DSOM error code. (DSOM error codes are listed in Appendix A of the *SOM Programming Guide*.)

Remarks

The **get_next_response** function returns a pointer to the next **Request** object to complete after starting multiple requests in parallel. This is a synchronization function used in conjunction with the **send_multiple_requests** function. There is no specific order in which requests will complete.

If the *response_flags* field is set to 0, this function will not return until the next request completion. If the caller does not want to become blocked, the RESP_NO_WAIT flag should be specified.

get_next_response

Related Information

Functions

- **send_multiple_requests**

Methods

- **send**
- **get_response**
- **invoke**

Example Code

See the example for the **send_multiple_requests** function.

ORBfree

Frees the memory of return values and **out** arguments that was allocated by DSOM on behalf of the caller of a remote method.

Syntax

```
void ORBfree  (void  *ptr)
```

Parameters

ptr (void *)
A pointer to memory that has been dynamically allocated by DSOM for a method return value or **out** argument.

Returns

rc (void)

Remarks

The **ORBfree** function is used to free memory for method return values or **out** arguments which are placed in memory allocated by DSOM (versus the calling program) on behalf of the *caller* of a remote method. For example, strings, arrays, sequence buffers, and "any" values are returned in memory which is dynamically-allocated by DSOM.

By contrast, memory that DSOM allocates on behalf of the *receiver* of a remote method (such as, for an object-owned **in** parameter) should be freed using the SOM kernel's **SOMFree** function rather than **ORBfree**.

This function is described in section 5.16, "Argument Passing Considerations", and section 5.17, "Return Result Passing Considerations", of the CORBA 1.1 specification.

Related Information

Functions

- **SOMD_NoORBfree**
- **SOMFree** (in SOM kernel)

Example Code

```
#include  <somd.h>
#include  <myobject.h>     /* provided by user */

MyObject obj;
Environment ev;
string str;

/* assume myMethod has the following IDL declaration
* in the MyObject interface:
*
*    void myMethod(out string s);
*/
_myMethod(obj, &ev, &str);
...
/* free storage */
ORBfree(str);
```

send_multiple_requests

This function initiates multiple **Requests** in parallel.

Syntax

```
ORBStatus send_multiple_requests (Request  reqs, Environment  *env,
                long  count, Flags  invoke_flags)
```

Parameters

reqs (Request)
The address of an array of **Requests** objects which are to be initiated in parallel.

env (Environment *)
A pointer to the **Environment** structure for the caller.

count (long)
The number of **Request** objects in *reqs*.

invoke_flags (Flags)
A **Flags** (unsigned long) value, used to indicate the following options:

INV_NO_RESPONSE	Indicates the caller does not intend to get any results or **out** parameter values from any of the requests. The requests can be treated as if they are **oneway** operations.
INV_TERM_ON_ERR	If one of the requests causes an error, the remaining requests are not sent.

The above flag values may be "or"-ed together.

Returns

rc (ORBStatus)
May return a non-zero **ORBStatus** value, which indicates a DSOM error code. (DSOM error codes are listed in Appendix A of the *SOM Programming Guide*.)

Remarks

The **send_multiple_requests** function initiates multiple **Requests** "in parallel". (The actual degree of parallelism is system dependent.) Each **Request** object is created using the **create_request** method, defined on **SOMDClientProxy**. Like the **send** method, this function returns to the caller immediately without waiting for the

send_multiple_requests

Requests to finish. The caller waits for the request responses using the **get_next_response** function.

This function is described in section 6.3, “Deferred Synchronous Routines”, of the CORBA 1.1 specification.

Related Information

Functions

- **get_next_response**

Methods

- **send**
- **get_response**
- **invoke**

Example Code

```
#include  <somd.h>

/* sum a set of values in parallel */
int parallel_sum(Environment *ev, int n, SOMDObject *objs)
{
   int index, sum = 0;
   Request *next;
   Request *reqs = (Request*) SOMMalloc(n *sizeof(Request));
   NamedValue *results = (NamedValue*)
                       SOMMalloc(n * sizeof(Namedvalue));

   for (i=0; i < n; i++)
      (void) _create_request((Context *)NULL, "_get_count", NULL,
                          &(result[i]), &(reqs[i]), (Flags)0);

   (void)  send_multiple_requests(reqs, ev, n, (Flags)0);

   for (i=0, i < n; i++)  {
      (void)  get_next_response(ev, (Flags)0, &next);
      index = (next - reqs);
      sum += *((int*)results[index].argument._value);
}

   return(sum);
}
```

somdExceptionFree

Frees the memory held by the exception structure within an **Environment** structure, regardless of whether the exception was returned by a local or a remote method call.

Syntax

```
void somdExceptionFree  (Environment  *env)
```

Parameters

env (Environment *)
The **Environment** structure whose exception information is to be freed.

Returns

rc (void)

Remarks

The **somdExceptionFree** function frees the memory held by the exception structure within an **Environment** structure, regardless of whether the exception was returned by a local or a remote method call.

When a DSOM client program invokes a remote method and the method returns an exception in the **Environment** structure, it is the client's responsibility to free the exception. This is done by calling either **exception_free** or **somdExceptionFree** on the **Environment** structure in which the exception was returned. (The two functions are equivalent. The **exception_free** function name is #defined in the som.h or som.xh file to provide strict CORBA compliance of function names.) There is a similar function, **somExceptionFree**, available for SOM programmers; DSOM programmers, however, can use **somdExceptionFree** to free all exceptions (regardless of whether they were returned from a local or a remote method call).

Related Information

Functions

- **somExceptionFree**
- **somExceptionID**
- **somExceptionValue**

- **somSetExceptionException**
- All SOM Kernel Functions

Data Structures

- **Environment**

Example Code

```
X_foo(x, ev, 23);  /* make a remote method call */
if (ev->major != NO_EXCEPTION)
{
    printf("foo exception = %s\n", somExceptionId(ev));

    /* ... handle exception ... */

    somdExceptionFree(ev);  /* free exception */
}
```

SOMD_Init

This function initializes DSOM in the calling process.

Syntax

```
void SOMD_Init  (Environment  *env)
```

Parameters

env (Environment *)
A pointer to the **Environment** structure for the method caller.

Returns

rc (void)

Remarks

Initializes DSOM in the calling process. This function should be called before any other DSOM functions or methods. This function should only be invoked (a) at the beginning of a DSOM program (client or server), to initialize the program, or (b) after **SOMD_Uninit** has been invoked, to reinitialize the program. If the program has already been initialized with **SOMD_Init**, then invoking **SOMD_Init** again has no effect.

An effect of calling **SOMD_Init** is that the global variables **SOMD_ObjectMgr, SOMD_ImplRepObject,** and **SOMD_ORBObject**, are initialized with pointers to the (single) instances of the **SOMDObjectMgr, ImplRepository,** and **ORB** objects.

The global variables **SOMD_ObjectMgr, SOMD_ImplRepObject,** and **SOMD_ORBObject** are set implicitly.

See Chapter 6 on DSOM in the *SOM Programming Guide*.

Example Code

```
#include  <somd.h>

Environment ev;

/* initialize Environment */
SOM_InitEnvironment(&ev);

/* initialize DSOM runtime */
SOMD_Init(&ev);
...

/* Free DSOM resources */
SOMD_Uninit(&ev);
```

SOMD_NoORBfree

This function specifies to DSOM that the client program will use the **SOMFree** function to free memory allocated by DSOM, rather than using the **ORBfree** function.

Syntax

```
void SOMD_NoORBfree  ()
```

Parameters

None.

Returns

rc (void)

Remarks

The **SOMD_NoORBfree** function is used in a DSOM client program to specify to DSOM that the client program will use the **SOMFree** function to free memory allocated by DSOM, rather than using the **ORBfree** function.

Typically, a DSOM client program will use **SOMFree** to free memory returned from local method calls and **ORBfree** to free memory returned from remote method calls. The **SOMD_NoORBfree** function allows programmers to use a single function (**SOMFree**) to free blocks of memory, regardless of whether they were allocated locally or by DSOM in response to a remote method call.

SOMD_NoORBfree, if used, should be called just after calling **SOMD_Init** in the client program. In response to this call, DSOM will not keep track of the memory it allocates for the client. Instead, it will assume that the client program will be responsible for walking all data structures returned from remote method calls, calling **SOMFree** for each block of memory within.

Related Information

Functions

- **ORBfree**

SOMD_NoORBfree

- **SOMFree**

Example Code

```
SOMD_Init();
SOMD_NoORBfree();

/* rest of client program */
```

SOMD_RegisterCallback

This function registers a callback function for handling DSOM request events.

Syntax

```
void SOMD_RegisterCallback  (SOMEEMan  emanObj,
                             EMRegProc  *func)
```

Parameters

emanObj (SOMEEMan)
A pointer to an instance of **SOMEEman**, the Event Manager object.

func (EMRegProc *)
A pointer to an event handler function which will be called by EMan whenever a DSOM request arrives. This function must have the following prototype (equivalent to the **EMRegProc** type defined in "eman.h"):

```
#ifdef __OS2__
 #pragma linkage( func, system)
#endif


void SOMLINK func (SOMEEvent event, void *eventData)
/* On Windows, using the SOMLINK keywork precludes
 * the support of multiple instances. */
```

Returns

rc (void)

Remarks

When writing event-driven applications where there are event sources other than DSOM requests (e.g., user input, mouse clicks, etc.), DSOM cannot be given exclusive control of the "main loop," such as when **execute_request_loop** is called. Instead, the application should use the Event Manager (EMan) framework to register and process all application events.

The **SOMD_RegisterCallback** function is used to register a user-supplied DSOM event handler function with EMan. The caller need only supply an address of the

event handler function, and the instance of the EMan object -- the details of registration are implemented by **SOMD_RegisterCallback**.

Callback functions should have the SOMLINK keyword explicitly specified, except on Windows. Using an explicit SOMLINK keyword on Windows will preclude the ability of an application to support multiple instances.

Note: The function **SOMD_RegisterCallback** must be declared with "system linkage" on OS/2.

See Chapter 9 of the *SOM Programming Guide* for a description of the Event Manager (EMan) framework, for writing event-driven applications.

Example Code

```
#include  <somd.h>
#include  <eman.h>

#ifdef __OS2__
 #pragma linkage(SOMD*RegisterCallback, system)
 #pragma linkage(DSOMEventCallBack, system)
#endif

/* On Windows, this example would omit the SOMLINK keyword. */

void SOMLINK DSOMEventCallBack (SOMEEvent event, void *eventData)
{
   Environment ev;
   SOM_InitEnvironment(&ev);

   _execute_request_loop (SOMD_SOMOAObject, &ev, SOMD_NO_WAIT);
}

main()
{

   ...
   eman = SOMEEmanNew();
   SOMD_RegisterCallback(eman, DSOMEventCallBack);

   _someProcessEvents(eman, &ev);   /* main loop */
   ...

}
```

SOMD_Uninit

This function frees system resources allocated for use by DSOM.

Syntax

```
void SOMD_Uninit  (Environment  *env)
```

Parameters

env (Environment *)
A pointer to the **Environment** structure for the method caller.

Returns

rc (void)

Remarks

Frees system resources (shared memory segments, semaphores, etc.) allocated to the calling process for use by DSOM. This function should be called before a process exits, to ensure system resources are reused.

No DSOM functions or methods should be called after **SOMD_Uninit** has been called. After **SOMD_Uninit** is called, the program can be reinitialized by calling **SOMD_Init**. (**SOMD_Uninit** would then need to be called again before program termination, to uninitialize the program.)

See Chapter 6 on DSOM in the *SOM Programming Guide*.

SOMD_Uninit

Example Code

```
#include  <somd.h>

Environment ev;

/* initialize Environment */
SOM_InitEnvironment(&ev);

/* initialize DSOM runtime */
SOMD_Init(&ev);
...
/* Free DSOM resources */
SOMD_Uninit(&ev);
```

Context_delete Macro

This macro deletes a **Context** object.

Syntax

```
ORBStatus Context_delete Macro (Context ctxobj, Environment *env,
                                Flags del_flag)
```

Parameters

ctxobj (Context)
A pointer to the **Context** object to be deleted.

env (Environment *)
A pointer to the **Environment** structure for the caller.

del_flag (Flags)
A bitmask (unsigned long). If the flag CTX_DELETE_DESCENDANTS is specified, the macro deletes the specified **Context** object and all of its descendant **Context** objects. A zero value indicates that the flag is not set.

Returns

rc (ORBStatus)

Remarks

This macro deletes the specified **Context** object. This macro maps to the **destroy** method of the **Context** class.

Expansion

Context_destroy (ctxobj, env, del_flag)

Related Information

Methods

- **Context_destroy**

Example Code

```
#include <somd.h>

Environment ev;
Context cxt, newcxt;
long rc;
...
/* get the process' default Context */
rc = _get_default_context(SOMD_ORBObject, &ev, &cxt);

/* make newcxt a child Context of the default Context (cxt) */
rc = _create_child(cxt, &ev, "myContext", &newcxt);
...
/* assuming no descendent Contexts have been
 * created from newcxt, we can destroy newcxt with flags=0
 */
rc = Context_delete(newcxt, &ev, (Flags) 0);
```

Request_delete Macro

This macro deletes the memory allocated by the ORB for a **Request** object.

Syntax

```
ORBStatus Request_delete Macro (Request reqobj, Environment *env)
```

Parameters

reqobj (Request)
A pointer to the **Request** object to be deleted.

env (Environment *)
A pointer to the **Environment** structure for the caller.

Returns

rc (ORBStatus)

Remarks

This macro deletes the specified **Request** object and all associated memory. This macro maps to the **destroy** method of the **Request** class.

Expansion

Request_destroy (reqobj, env)

Related Information

Methods

- **Request_destroy**

Example Code

```
#include <somd.h>
#include <repostry.h>
#include <intfacdf.h>
#include <foo.h>   /* provided by user */

/* assume following method declaration in interface Foo:
 *    long methodLong  (in long inLong,inout long inoutLong);
 * then the following code sends a request to execute the call:
 *  result = methodLong(fooObj, &ev, 100,200);
 * using the DII without waiting for the result. Then, later,
 * waits for and then uses the result.
 */
Environment ev;
NVList arglist;
long rc;
Foo fooObj;
Request reqObj;
NamedValue result;

/* see the Example code for invoke to see how the request
 * is built
 */

/* Create the Request, reqObj */
rc = _create_request(fooObj, &ev, (Context *)NULL, "methodLong",
                     arglist, &result, &reqObj, (Flags)0);

/* Finally, send the request */
rc = _send(reqObj, &ev, (Flags)0);

/* do some work, i.e. don't wait for the result */

/* wait here for the result of the request */
rc = _get_response(reqObj, &ev, (Flags)0);

/* use the result */
if (result->argument._value == 9600) {...}

/* throw away the reqObj */
Request_delete(reqObj, &ev);
```

BOA

File stem: boa

Base

SOMObject

Metaclass

SOMMSingleInstance

Ancestor Class

SOMObject

Subclass

SOMOA

Description

The Basic Object Adapter (**BOA**) defines the basic interfaces that a server process uses to access services of an Object Request Broker like DSOM. The **BOA** defines methods for creating and exporting object references, registering implementations, activating implementations and authenticating requests.

For more information on the Basic Object Adapter, refer to Chapter 9 in the CORBA 1.1 specification.

Note: DSOM treats the **BOA** interface as an *abstract* class, which merely defines basic runtime interfaces (introduced in the CORBA specification) but does not implement those interfaces. Thus, there is no point in instantiating a **BOA** object. If a **BOA** object is created, any methods invoked on it will return a NO_IMPLEMENT exception. Instead, the SOM Object Adapter (**SOMOA**) subclass provides DSOM implementations for **BOA** methods. When a **BOA** method is invoked on the **SOMOA** object, the desired behavior will occur.

New methods

The following list shows all the BOA methods.

- change_implementation
- create
- deactivate_impl
- deactivate_obj
- dispose
- get_id
- get_principal

- impl_is_ready
- obj_is_ready
- set_exception

Overridden methods

There are currently no overridden methods defined for the BOA class.

change_implementation

This method changes the implementation associated with the referenced object. *(Not implemented.)*

Syntax

```
void change_implementation (BOA receiver, Environment *env,
                            SOMDObject obj, ImplementationDef impl)
```

Parameters

receiver (BOA)
A pointer to a **BOA** (**SOMOA**) object for the server.

env (Environment *)
A pointer to the **Environment** structure for the method caller.

obj (SOMDObject)
A pointer to the **SOMDObject** object which refers to the application object whose implementation is to be changed.

impl (ImplementationDef)
A pointer to the **ImplementationDef** object representing the new implementation of the application object.

Returns

rc (void)
The **SOMOA** implementation always returns a NO_IMPLEMENT exception, with a minor code of SOMDERROR_NotImplemented.

Remarks

The **change_implementation** method is defined by the CORBA specification, *but has a null implementation in DSOM.* This method always returns a NO_IMPLEMENT exception.

In CORBA 1.1, the **change_implementation** method is provided to allow an application to change the implementation definition of an object.

However, in DSOM, the **ImplementationDef** identifies the server which implements an object. In these terms, changing an object's implementation (i.e., server) would

result in a change in the object's location. In DSOM, moving objects from one server to another is considered an application-specific task, and hence, no default implementation is provided.

It *is* possible, however, to change the program which implements an object's server, or change the class library which implements an object's class. To modify the program associated with an **ImplementationDef**, use the **update_impldef** method defined on **ImplRepository.** To change the implementation of an object's class, replace the corresponding class library with a new (upward-compatible) one.

Original Class

BOA

create

This method creates a "reference" for a local application object which can be exported to remote clients.

Syntax

```
SOMDObject create  (BOA  receiver, Environment  *env,
                    ReferenceData  id, InterfaceDef  intf,
                    ImplementationDef  impl)
```

Parameters

receiver (BOA)
A pointer to a **BOA** (**SOMOA**) object for the server.

env (Environment *)
A pointer to the **Environment** structure for the method caller.

id (ReferenceData)
A pointer to the **ReferenceData** structure containing application-specific information describing the target object.

intf (InterfaceDef)
A pointer to the **InterfaceDef** object which describes the interface of the target object.

impl (ImplementationDef)
A pointer to the **ImplementationDef** object which describes the application (server) process which implements the target object.

Returns

rc (SOMDObject)
Returns a pointer to a **SOMDObject** which refers to a local application object.

Remarks

The **create** method creates a **SOMDObject** which is used as a "reference" to a local application object. An object reference is simply an object which is used to refer to another target object -- one may think of it as an "id", "link", or "handle." Object references are important in DSOM in that their values can be externalized (i.e., can be

represented in a string form) for transmission between processes, storage in files, and so on. In DSOM, the proxy objects in client processes are remote object references.

File somdtype.idl contains the specification: **typedef sequence<octet,1024> ReferenceData**. To create an object reference, the caller specifies the **ImplementationDef** of the calling process, the **InterfaceDef** of the target application object, and up to 1024 bytes of **ReferenceData** which is used by the application to identify and activate the application object. When subsequent method calls specify the object reference as a parameter, the application will use the reference to find and/or activate the referenced object.

Note that (as specified in CORBA 1.1) each call to **create** returns a unique object reference, even if the same parameters are used in subsequent calls. For each reference, the **ReferenceData** is stored in the reference data file (and backup file, if any) for the server.

The **SOMOA** class introduces a **change_id** method which allows a server to modify the **ReferenceData** of one of its references. (The **change_id** method is *not* in the CORBA 1.1 specification.)

Ownership of the returned **SOMDObject**is transferred to the caller.

Original Class

BOA

Related Methods

Methods

- **change_id**
- **create_constant**
- **create_SOM_ref**
- **dispose**
- **get_id**

Example Code

```
#include  <somd.h>
#include  <repostry.h>
#include  <intfacdf.h>

Environment ev;
ReferenceData id;
InterfaceDef intfdef;
SOMDObject objref;
string fname; /* a file name to be saved with reference */
...
/* create the id for the reference */
id._maximum = id._length = strlen(fname)+1;
id._buffer = (string) SOMMalloc(strlen(fname)+1);
strcpy(id._buffer,fname);

/* get the interface def object for interface Foo*/
intfdef = _lookup_id(SOM_InterfaceRepository, &ev, "Foo");

objref =  _create(SOMD_SOMOAObject,
               &ev, id, intfdef, SOMD_ImplDefObject);
...
```

deactivate_impl

This method indicates that a server implementation is no longer ready to process requests.

Syntax

```
void deactivate_impl  (BOA  receiver, Environment  *env,
                       ImplementationDef  impl)
```

Parameters

receiver (BOA)
A pointer to a **BOA** (**SOMOA**) object for the server.

env (Environment *)
A pointer to the **Environment** structure for the method caller.

impl (ImplementationDef)
A pointer to the **ImplementationDef** object representing the implementation to be deactivated.

Returns

rc (void)

Remarks

The **deactivate_impl** method indicates that the implementation is no longer ready to process requests.

Original Class

BOA

Related Methods

Methods

- **impl_is_ready**
- **activate_imple_failed**
- **execute_request_loop**
- **execute_next_request**

Example Code

```
#include  <somd.h>

ORBStatus rc;

/* server initialization code ... */

/* signal DSOM that server is ready */
_impl_is_ready(SOMD_SOMOAObject, &ev, SOMD_ImplDefObject);

for(rc = 0;rc==0;) {
    rc = _execute_next_request(SOMD_SOMOAObject, &ev, waitFlag);
    /* perform app specific code between messages here, e.g.,*/
    numMessagesProcessed++;
}

/* signal DSOM that server is deactivated */
_deactivate_impl(SOMD_SOMOAObject, &ev, SOMD_ImplDefObject);
```

deactivate_obj

This method indicates that an object server is no longer ready to process requests. *(Not implemented.)*

Syntax

```
void deactivate_obj  (BOA  receiver, Environment  *env,
                      SOMDObject  obj)
```

Parameters

receiver (BOA)
A pointer to a **BOA** (**SOMOA**) object for the server.

env (Environment *)
A pointer to the **Environment** structure for the method caller.

obj (SOMDObject)
A pointer to a **SOMDObject** which identifies the object (server) to be deactivated.

Returns

rc (void)

Remarks

The **deactivate_obj** method is defined by the CORBA specification, *but has a null implementation in DSOM*. This method always returns a NO_IMPLEMENT exception.

CORBA 1.1 distinguishes between servers that implement many objects (“shared”), versus qervers that implement a single object (“unshared”). The **deactivate_obj** method is meant to be used by unshared servers, to indicate that the object (i.e., server) is no longer ready to process requests.

DSOM does not distinguish between servers that implement a single object versus servers that implement multiple objects, so this method has no implementation.

Original Class

BOA

Related Methods

Methods

- **deactivate_impl**
- **impl_is_ready**
- **obj_is_ready**

dispose

This method destroys an object reference.

Syntax

```
void dispose  (BOA  receiver, Environment  *env, SOMDObject  obj)
```

Parameters

receiver (BOA)
A pointer to a **BOA** object for the server.

env (Environment *)
A pointer to the **Environment** structure for the method caller.

obj (SOMDObject)
A pointer to the object reference to be destroyed.

Returns

rc (void)

Remarks

The **dispose** method disposes of an object reference.

Original Class

BOA

Related Methods

Methods

- **create**
- **create_constant**
- **create_SOM_ref**
- **get_id**

Example Code

```
#include  <somd.h>
#include  <repostry.h>
#include  <intfacdf.h>

SOMDObject objref;
ReferenceData id;
InterfaceDef intfdef;
...
objref =
 _create(SOMD_SOMOAObject, &ev, id, intfdef, SOMD_ImplDefObject);
...
_dispose(SOMD_SOMOAObject, &ev, objref);
```

get_id

This method returns reference data associated with the referenced object.

Syntax

```
ReferenceData get_id  (BOA  receiver, Environment  *env,
                       SOMDObject  obj)
```

Parameters

receiver (BOA)
A pointer to a **BOA** (**SOMOA**) object for the server.

env (Environment *)
A pointer to the **Environment** structure for the method caller.

obj (SOMDObject)
A pointer to a **SOMDObject** object for which to return the **ReferenceData**.

Returns

rc (ReferenceData)
Returns a **ReferenceData** structure associated with the referenced object.

Remarks

The **get_id** method returns the reference data associated with the referenced object.

Original Class

BOA

Related Methods

Methods

- **create**
- **create_constant**
- **dispose**

Example Code

```
#include  <somd.h>
#include  <repostry.h>
#include  <intfacdf.h>

SOMDObject objref;
ReferenceData id1, id2;
InterfaceDef intfdef;
...
objref =
 _create(SOMD_SOMOAObject,&ev, id1, intfdef, SOMD_ImplDefObject);
...
/* get the ReferenceData from a SOMDObject */
id2 =  _get_id(SOMD_SOMOAObject, &ev, objref);
```

get_principal

This method returns the ID of the principal that issued the request.

Syntax

```
Principal get_principal  (BOA  receiver, Environment  *env,
                          SOMDObject  obj, Environment  *req_ev)
```

Parameters

receiver (BOA)
A pointer to a **BOA** (**SOMOA**) object for the server.

env (Environment *)
A pointer to the **Environment** structure for the method caller.

obj (SOMDObject)
A pointer to the object reference which is the target of the method call.

req_ev (Environment *)
A pointer to the **Environment** object passed as input to the request.

Returns

rc (Principal)
Returns a pointer to a **Principal** object which identifies the user and host from which a request originated.

Remarks

The **get_principal** method returns the ID of the principal that issued a request.

Original Class

BOA

Related Methods

Classes

- **Principal**

Example Code

```
#include  <somd.h>

/* assumed context: inside a method implementation */
void methodBody(SOMObject *somSelf, Environment *ev, ...)
{
   Principal p;
   SOMDObject selfRef;
   Environment localev;

   SOMInitEnvironment(&localev);

   /* get a reference to myself from the server object */
   selfRef =
     somdRefFromSOMObj(SOMD_ServerObject, &ev, somSelf);

   /* get principal information from the SOMOA */
   p =  _get_principal(SOMD_SOMOAObject, &localev, selfRef, ev);

   printf("user = %s, host = %s\n",
         __get_userName(p), __get_hostName(p));
   ...
}
```

impl_is_ready

This method indicates that the implementation is ready to process requests.

Syntax

```
void impl_is_ready  (BOA  receiver, Environment  *env,
                     ImplementationDef  impl)
```

Parameters

receiver (BOA)
A pointer to a **BOA** (**SOMOA**) object for the server.

env (Environment *)
A pointer to the **Environment** structure for the method caller.

impl (ImplementationDef)
A pointer to the **ImplementationDef** object indicating which implementation is ready.

Returns

rc (void)

Remarks

The **impl_is_ready** method Indicates that the implementation is ready to process requests.

Original Class

BOA

Related Methods

Methods

- **deactivate_impl**
- **activate_impl_failed**
- **obj_is_ready**
- **execute_request_loop**
- **execute_next_request**

Example Code

```
#include  <somd.h>   /* needed by all servers */

main(int argc, char **argv)
{
   Environment ev;
   SOM_InitEnvironment(&ev);

   /* Initialize the DSOM run-time environment */
   SOMD_Init(&ev);

   /* Retrieve its ImplementationDef from the Implementation
      Repository by passing its implementation ID as a key */
   SOMD_ImplDefObject =
      _find_impldef(SOMD_ImplRepObject, &ev, argv[1]);

   /* Tell DSOM that the server is ready to process requests */
   _impl_is_ready(SOMD_SOMOAObject, &ev, SOMD_ImplDefObject);
   ...
}
```

obj_is_ready

Indicates that an object (server) is ready to process requests. *(Not implemented.)*

Syntax

```
void obj_is_ready (BOA receiver, Environment *env, SOMDObject obj,
                   ImplementationDef impl)
```

Parameters

receiver (BOA)
A pointer to a **BOA** (**SOMOA**) object for the server.

env (Environment *)
A pointer to the **Environment** structure for the method caller.

obj (SOMDObject)
A pointer to a **SOMDObject** which identifies the object (server) which is ready.

impl (ImplementationDef)
A pointer to the **ImplementationDef** object representing the object that is ready.

Returns

rc (void)

Remarks

The **obj_is_ready** method is defined by the CORBA specification, *but has a null implementation in DSOM.* This method always returns a NO_IMPLEMENT exception.

CORBA 1.1 distinguishes between servers that implement many objects (“shared”), versus servers that implement a single object (“unshared”). The **obj_is_ready** method is meant to be used by unshared servers, to indicate that the object (i.e., server) is ready to process requests.

DSOM does not distinguish between servers that implement a single object versus servers that implement multiple objects, so this method has no implementation.

Original Class

BOA

Related Methods

Methods

- **impl_is_ready**
- **deactivate_impl**
- **deactivate_obj**
- **activate_impl_failed**

set_exception

This method returns an exception to a client.

Syntax

```
void set_exception  (BOA  receiver, Environment  *env,
                     exception_type  major, string  except_name,
                     void  *param)
```

Parameters

receiver (BOA)
A pointer to a **BOA** (**SOMOA**) object for the server.

env (Environment *)
A pointer to the **Environment** structure for the method caller.

major (exception_type)
One of the exception types NO_EXCEPTION, USER_EXCEPTION, or SYSTEM_EXCEPTION.

except_name (string)
A **string** representing the exception type identifier.

param (void *)
A pointer to the associated data.

Returns

rc (void)

Remarks

The **set_exception** method returns an exception to the client. The *major* parameter can have one of three possible values:

NO_EXCEPTION Indicates a normal outcome of the operation. It is not necessary to invoke **set_exception** to indicate a normal outcome; it is the default behavior if the method simply returns.

USER_EXCEPTION Indicates a user-defined exception.

SYSTEM_EXCEPTION Indicates a system-defined exception.

Original Class

BOA

Example Code

```
#include  <somd.h>
#include  <myobject.h>    /* provided by user */

/* assuming following IDL declarations in the MyObject interface:
 *    exception foo;
 *    void myMethod() raises (BadCall);
 * then within the implementation of myMethod, the
 * following call can raise a BadCall exception: */

_set_exception(SOMD_SOMOAObject,
               &ev, USER_EXCEPTION, ex_MyObject_BadCall, NULL);
```

Context

File stem: cntxt

Base

SOMObject

Metaclass

SOMClass

Ancestor Classes

SOMObject

Description

The **Context** class implements the CORBA Context object described in section 6.5 beginning on page 116 of CORBA 1.1. A **Context** object contains a list of properties, each consisting of a name and a string value associated with that name. **Context** objects are created/accessed by the **get_default_context** method defined in the **ORB** object.

New methods

The following list shows all the Context methods.

- create_child
- delete_values
- destroy*
- get_values
- set_one_value
- set_values

(***** The **destroy** method was defined as **delete** in CORBA 1.1, which conflicts with the **delete** operator in C++. However, there is a **Context_delete** macro defined for CORBA compatibility.)

Overridden methods

The following list shows all the methods overridden by the Context class. These methods are overridden in order to modify the behavior defined by an ancestor class.

- somInit

create_child

This method creates a child of a **Context** object.

Syntax

```
ORBStatus create_child  (Context  receiver, Environment  *env,
                         Identifier  ctx_name, Context  child_ctx)
```

Parameters

receiver (Context)
A pointer to the **Context** object for which a child is to be created.

env (Environment *)
A pointer to the **Environment** structure for the method caller.

ctx_name (Identifier)
The name of the child **Context** to be created.

child_ctx (Context)
The address where a pointer to the created child **Context** object is to be stored.

Returns

rc (ORBStatus)
Returns an **ORBStatus** value representing the return code from the operation.

Remarks

The **create_child** method creates a child **Context** object.

The returned **Context** object is chained to its parent. That is, searches on the child **Context** object will look in the parent (and so on, up the **Context**tree), if necessary, for matching property names.

Original Class

Context

Example Code

```
#include  <somd.h>

Environment ev;
Context cxt, newcxt;
long rc;
...
/* get the process' default Context */
rc = _get_default_context(SOMD_ORBObject, &ev, &cxt);
/* make newcxt a child Context of the default Context (cxt) */
rc =  _create_child(cxt, &ev, "myContext", &newcxt);
```

delete_values

This method deletes property value(s).

Syntax

```
ORBStatus delete_values  (Context  receiver, Environment  *env,
                          Identifier  prop_name)
```

Parameters

receiver (Context)
A pointer to the **Context** object from which values will be deleted.

env (Environment *)
A pointer to the **Environment** structure for the method caller.

prop_name (Identifier)
An identifier specifying the property value(s) to be deleted.

Returns

rc (ORBStatus)
Returns an **ORBStatus** value representing the return code from the operation.

Remarks

The **delete_values** method deletes the specified property value(s) from a **Context** object. If *prop_name* has a trailing wildcard character("*"), then all property names that match will be deleted.

Search scope is always limited to the specified **Context** object.

If no matching property is found, an exception is returned.

Original Class

Context

Related Methods

Methods

- **set_one_value**

delete_values

- **set_values**
- **get_values**

Example Code

```
#include  <somd.h>

Environment ev;
Context cxt, newcxt;
long rc;
...
/* get the process' default Context */
rc = _get_default_context(SOMD_ORBObject, &ev, &cxt);
/* make newcxt a child Context of the default Context (cxt) */
rc = _create_child(cxt, &ev, "myContext", &newcxt);
rc = _set_one_value(newcxt, &ev, "username", "joe");
...
rc =  _delete_values(newcxt, &ev, "username");
```

destroy

This method deletes a **Context** object.

Syntax

```
ORBStatus destroy  (Context  receiver, Environment  *env, Flags  del_flag)
```

Parameters

receiver (Context)
A pointer to the **Context** object to be deleted.

env (Environment *)
A pointer to the **Environment** structure for the method caller.

del_flag (Flags)
A bitmask (unsigned long). If the option flag **CTX_DELETE_DESCENDENTS** is specified, the method deletes the indicated **Context** object and all of its descendent **Context** objects. Or, a zero value indicates the flag is not set.

Returns

rc (ORBStatus)
Returns an **ORBStatus** value representing the return code from the operation.

Remarks

The **destroy** method deletes the specified **Context** object.

Note: This method is called “delete” in the CORBA 1.1 specification. However, the word “delete” is a reserved operator in C++, so the name “destroy”was chosen as an alternative. For CORBA compatibility, a macro defining **Context_delete** as an alias for **destroy** has been included in the C header files.

Original Class

Context

destroy

Example Code

```
#include  <somd.h>

Environment ev;
Context cxt, newcxt;
long rc;
...
/* get the process' default Context */
rc = _get_default_context(SOMD_ORBObject, &ev, &cxt);
/* make newcxt a child Context of the default Context (cxt) */
rc =  _create_child(cxt, &ev, "myContext", &newcxt);
...
/* assuming no descendent Contexts have been
 * created from newcxt, we can destroy newcxt with flags=0
 */
rc =  _destroy(newcxt, &ev, (Flags) 0);
```

get_values

This method retrieves the specified property values.

Syntax

```
ORBStatus get_values  (Context  receiver, Environment  *env,
                       Identifier  start_scope, Flags  op_flags,
                       Identifier  prop_name, NVList  values)
```

Parameters

receiver (Context)
A pointer to the **Context** object from which the properties are to be retrieved.

env (Environment *)
A pointer to the **Environment** structure for the method caller.

start_scope (Identifier)
An **Identifier** specifying the name of the **Context** object at which search for the properties should commence.

op_flags (Flags)
This parameter is currently not used (the CTX_RESTRICT_SCOPE flag is currently not supported); callers can simply pass zero. object.

prop_name (Identifier)
An **Identifier** specifying the name of the property value(s) to return.

values (NVList)
The address to store a pointer to the resulting **NVList** object.

Returns

rc (ORBStatus)
Returns an **ORBStatus** value representing the return code from the operation.

Remarks

The **get_values** method retrieves the specified **Context** property values(s). If *prop_name* has a trailing wildcard character("*"), then all matching properties and their values are returned. OWNERSHIP of the returned **NVList** object is transferred to the caller.

If no properties are found, an error is returned and no property list is returned.

Scope indicates the level at which to initiate the search for the specified properties. If a property is not found at the indicated level, the search continues up the **Context** object tree until a match is found or all **Context** objects in the chain have been exhausted.

If scope name is omitted, the search begins with the specified **Context** object. If the specified scope name is not found, an exception is returned.

Original Class

Context

Related Methods

Methods

- **set_one_value**
- **set_values**
- **delete_values**

Example Code

```
#include  <somd.h>

Environment ev;
Context cxt1, cxt2;
string *cxt1props;
long rc, i, numprops;
NVList nvp;
...
for (i= numprops; i > 0; i--) {
 /* get the value of the *cxt1props property from cxt1 */
 rc =  _get_values(cxt1, &ev, NULL, (Flags) 0, *cxt1props, &nvp);
 /* and if found then update cxt2 with that name-value pair */
 if (rc == 0) rc = _set_values(cxt2, &ev, nvp);
 _free(nvp,&ev);
 cxt1props++;
}
```

set_one_value

This method adds a single property to the specified **Context** object.

Syntax

```
ORBStatus set_one_value  (Context  receiver, Environment  *env,
                          Identifier  prop_name, string  value)
```

Parameters

receiver (Context)
A pointer to the **Context** object to which the value is to be added.

env (Environment *)
A pointer to the **Environment** structure for the method caller.

prop_name (Identifier)
The name of the property to be added. The *prop_name* should not end in an asterisk (*).

value (string)
The value of the property to be added.

Returns

rc (ORBStatus)
Returns an **ORBStatus** value representing the return code from the operation.

Remarks

The **set_one_value** method adds a single property to the specified **Context** object.

Original Class

Context

Related Methods

Methods

- **set_values**
- **get_values**
- **delete_values**

set_one_value

Example Code

```
#include  <somd.h>
Environment ev;
Context cxt, newcxt;
long rc;

/* get the process' default Context */
rc = _get_default_context(SOMD_ORBObject, &ev, &cxt);
/* make newcxt a child Context of the default Context (cxt) */
rc = _create_child(cxt, &ev, "myContext", &newcxt);
rc = _set_one_value(newcxt, &ev, "username", "joe");
```

set_values

This method adds/changes one or more property values in the specified **Context** object.

Syntax

```
ORBStatus set_values  (Context  receiver, Environment  *env,
                       NVList  values)
```

Parameters

receiver (Context)
A pointer to the **Context** object for which the properties are to be set.

env (Environment *)
A pointer to the **Environment** structure for the method caller.

values (NVList)
A pointer to an **NVList** object containing the properties to be set.

Returns

rc (ORBStatus)
Returns an **ORBStatus** value representing the return code from the operation.

Remarks

The **set_values** method sets one or more property values in the specified **Context** object. In the **NVList**, the flags field must be set to zero and the TypeCode field associated with an attribute value must be TC_string.

Original Class

Context

Related Methods

Methods

- **set_one_value**
- **get_values**
- **delete_values**

set_values

Example Code

```
#include  <somd.h>

Environment ev;
Context cxt1, cxt2;
string *cxt1props;
long rc, i, numprops;
NVList nvp;
...
for (i= numprops; i > 0; i--) {
 /* get the value of the *cxt1props property from cxt1 */
 rc = _get_values(cxt1, &ev, NULL, (Flags) 0, *cxt1props, &nvp);
 /* and if found then update cxt2 with that name-value pair */
 if (rc == 0) rc =  _set_values(cxt2, &ev, nvp);
 _free(nvp,&ev);
 cxt1props++;
}
```

ImplementationDef

File stem: impldef

Base

SOMObject

Metaclass

SOMClass

Ancestor Classes

SOMObject

Description

The **ImplementationDef** class defines attributes necessary for the DSOM daemon to find and activate the implementation of an object.

Note: * Details of the **ImplementationDef** object are not currently defined in the CORBA 1.1 specification; the attributes which have been defined are required by DSOM.

Attributes

Listed below is each available attribute, with its corresponding type in parentheses, followed by a description of its purpose:

impl_id (string)
Contains the DSOM-generated identifier for a server implementation.

impl_alias (string)
Contains the "alias" (user-friendly name) for a server implementation.

impl_program (string)
Contains the name of the program or command file which will be executed when a process for this server is started automatically by **somdd**. If the full pathname is not specified, the directories specified in the PATH environment variable will be searched for the named program or command file.

Optionally, the server program can be run under control of a "shell" or debugger, by specifying the shell or debugger name first, followed by the name of the server program. (A space separates the two program names.) For example,

```
dbx myserver
```

Servers that are started automatically by **somdd** will always be passed their **impl_id** as the first parameter.

impl_flags (Flags)
Contains a bit-vector of flags used to identify server options. Currently, the IMPLDEF_MULTI_THREAD flag indicates that each request should be executed on a separate thread. IMPLDEF_DISABLE_SVR indicates that the server process has been disabled from starting. `IMPLDEF_NONSTOPPABLE` indicates that the server cannot be shutdown by using the **dsom stop** or **dsom restart** commands (see "The 'dsom' server manager utility" in Chapter 6 of the *SOM Programming Guide*) or by using the corresponding methods of the **SOMDServerMgr** class (**somdShutdownServer** and **somdRestartServer**). `IMPLDEF_IMPLID_SET` indicates that the **impl_id** attribute has already been set when the **ImplementationDef** object is passed to the **ImplRepository::add_impldef** method. The **ImplRepository::add_impldef** method normally generates the **impl_id** attribute before storing the **ImplementationDef** object in the Implementation Repository.

impl_server_class (string)
Contains the name of the **SOMDServer** class or subclass created by the server process.

impl_refdata_file (string)
Contains the full pathname of the file used to store **ReferenceData** for the server.

impl_refdata_bkup (string)
Contains the full pathname of the backup mirror file used to store **ReferenceData** for the server.

impl_hostname (string)
Contains the hostname of the machine where the server is located.

Note:

Currently, when stored in the Implementation Repository, file names used in **ImplementationDef**s are limited to 255 bytes. Implementations aliases used in **ImplementationDef**s are limited to 50 bytes. Class names used in **ImplementationDef**s are limited to 50 bytes. Hostnames are limited to 32 bytes.

New methods

There are currently no new methods defined for the ImplementationDef class.

Overridden methods

There are currently no overridden methods defined for the ImplementationDef class.

ImplRepository

File stem: implrep

Base

SOMObject

Metaclass

SOMMSingleInstance

Ancestor Classes

SOMObject

Description

The **ImplRepository** class defines operations necessary to query and update the DSOM Implementation Repository.

Note: * The Implementation Repository is described in concept in the CORBA 1.1 specification, but no standard interfaces have been defined. These interfaces have all been introduced by DSOM. In addition to using the following interfaces, the DSOM Implementation Repository can be queried and updated using the **regimpl** tool.

New methods

The following list shows all the ImplRepository methods.

- add_class_to_impldef
- add_impldef
- delete_impldef
- find_all_impldefs
- find_classes_by_impldef
- find_impldef
- find_impldef_by_alias
- find_impldef_by_class
- remove_class_from_all
- remove_class_from_impldef
- update_impldef

Overridden methods

The following list shows all the methods overridden by the ImplRepository class. These methods are overridden in order to modify the behavior defined by an ancestor class.

- somInit
- somUninit

add_class_to_impldef

This method associates a class with a server.

Syntax

```
void add_class_to_impldef  (ImplRepository  receiver, Environment  *env,
                            ImplId  implid, string  classname)
```

Parameters

receiver (ImplRepository)
A pointer to the **ImplRepository** object.

env (Environment *)
A pointer to the **Environment** structure for the method caller.

implid (ImplId)
The **ImplId** identifier for the **ImplementationDef** of the desired server.

classname (string)
A **string** identifying the class name.

Returns

rc (void)
An exception is returned if there was an error updating the Implementation Repository.

Remarks

Associates a class, identified by name, with a server, identified by its **ImplId**. This type of association is used to lookup server implementations via the **find_impldef_by_class** method.

Original Class

ImplRepository

Example Code

```
#include  <somd.h>

Environment ev;
SOMDServer server;
ImplementationDef impldef;
ImplId implid;
...
server = _somdFindServerByName(SOMD_ObjectMgr,&ev,"stackServer");
impldef = _get_implementation(server,&ev);
implid = __get_impl_id(impldef,&ev);
_add_class_to_impldef(SOMD_ImplRepObject,&ev,implid , "Queue");
```

add_impldef

This method adds an implementation definition to the Implementation Repository.

Syntax

```
void add_impldef  (ImplRepository  receiver, Environment  *env,
                   ImplementationDef  impldef)
```

Parameters

receiver (ImplRepository)
A pointer to the **ImplRepository** object.

env (Environment *)
A pointer to the **Environment** structure for the method caller.

impldef (ImplementationDef)
A pointer to the **ImplementationDef** object to add to the Implementation Repository.

Returns

rc (void)
An exception is returned if there was an error updating the Implementation Repository.

Remarks

Adds the specified **ImplementationDef** object to the Implementation Repository.

Note: Unless the **impl_flags** attribute of the given **ImplementationDef** object contains the `IMPLDEF_IMPLID_SET` flag, the **impl_id** attribute of the **ImplementationDef** object is ignored, and a new **impl_id** value is created for the newly added **ImplementationDef** object.

Original Class

ImplRepository

Example Code

```
#include  _somd.h>

Environment ev;
ImplementationDef impldef;
...
impldef = ImplementationDefNew();
__set_impl_program(impldef,&ev,"/u/servers/myserver");
/* set more of the impldef's attributes here */
...
_add_impldef(SOMD_ImplRepObject,&ev,impldef);
```

delete_impldef

This method deletes an implementation definition from the Implementation Repository.

Syntax

```
void delete_impldef  (ImplRepository  receiver, Environment  *env,
                      ImplId  implid)
```

Parameters

receiver (ImplRepository)
A pointer to the **ImplRepository** object.

env (Environment *)
A pointer to the **Environment** structure for the method caller.

implid (ImplId)
The **ImplId** that identifies the server implementation of interest.

Returns

rc (void)
An exception is returned if there was an error updating the Implementation Repository.

Remarks

Deletes the specified **ImplementationDef** object from the Implementation Repository.

Original Class

ImplRepository

Example Code

```
#include  <somd.h>

Environment ev;
ImplementationDef impldef;
...
impldef =

_find_impldef_by_name(SOMD_ImplRepObject,&ev,"stackServer");
_delete_impldef(SOMD_ImplRepObject,&ev,__get_impl _id(impldef,&ev));
```

find_all_impldefs

This method returns all the implementation definitions in the Implementation Repository.

Syntax

```
ORBStatus find_all_impldefs (ImplRepository  receiver,
                             Environment  *env,
                             sequence<ImplementationDefs>  outimpldefs)
```

Parameters

receiver (ImplRepository)
A pointer to an object of class **ImplRepository**.

env (Environment *)
A pointer to the **Environment** structure for the calling method.

outimpldefs (sequence<ImplementationDefs>)
A sequence of **ImplementationDefs** is returned.

Returns

rc (ORBStatus)
A zero is returned to indicate success; otherwise, a DSOM error code is returned.

Remarks

The **find_all_impldefs** method searches the Implementation Repository and returns all the **ImplementationDef** objects in it.

Original Class

ImplRepository

Example Code

```
#include <somd.h>

Environment ev;
sequence (ImplementationDef) impldefs;

...

find_all_impldefs(SOMD_ImplRepObject, &ev, &impldefs);
```

find_classes_by_impldef

This method returns a sequence of class names associated with a server.

Syntax

```
sequence<string> find_classes_by_impldef (ImplRepository  receiver,
                                          Environment  *env, ImplId  implid)
```

Parameters

receiver (ImplRepository)
A pointer to the **ImplRepository** object.

env (Environment *)
A pointer to the **Environment** structure for the method caller.

implid (ImplId)
The **ImplId** that identifies the server implementation of interest.

Returns

rc (sequence<string>)
A sequence of strings is returned. *Ownership* of the sequence structure, the string array buffer, and the strings themselves is transferred to the caller.

An exception is returned if there was an error reading the Implementation Repository.

Remarks

Searches the class index and returns the sequence of class names supported by a server with the specified *implid*.

Original Class

ImplRepository

Example Code

```
#include  <somd.h>

Environment ev;
SOMDServer server;
ImplementationDef impldef;
ImplId implid;
sequence(string) classes;
...
server = _find_server_by_name(SOMD_ObjectMgr,&ev,"stackServer");
impldef = _get_implementation(server,&ev);
implid = __get_impl_id(impldef,&ev);
classes = _find_classes_by_impldef(SOMD_ImplRepObject,&ev,implid);
```

find_impldef

This method returns a server implementation definition given its ID.

Syntax

```
ImplementationDef find_impldef  (ImplRepository  receiver,
                                 Environment  *env, ImplId  implid)
```

Parameters

receiver (ImplRepository)
A pointer to the **ImplRepository** object.

env (Environment *)
A pointer to the **Environment** structure for the method caller.

implid (ImplId)
The **ImplId** of the desired **ImplementationDef**.

Returns

rc (ImplementationDef)
A copy of the desired **ImplementationDef** object is returned. *Ownership* of the object is transferred to the caller.

An exception is returned if there was an error reading the Implementation Repository.

Remarks

Finds and returns the **ImplementationDef** object whose ID is *implid*.

Original Class

ImplRepository

Example Code

```
#include  <somd.h>

main(int argc, char **argv)
{
   Environment ev;
   SOM_InitEnvironment(&ev);

   /* Initialize the DSOM run-time environment */
   SOMD_Init(&ev);

   /* Retrieve its ImplementationDef from the Implementation
      Repository by passing its implementation ID as a key */
   SOMD_ImplDefObject =
      _find_impldef(SOMD_ImplRepObject, &ev, argv[1]);

   /* Tell DSOM that the server is ready to process requests */
   _impl_is_ready(SOMD_SOMOAObject, &ev, SOMD_ImplDefObject);
   ...
}
```

find_impldef_by_alias

This method returns a server implementation definition given its user-friendly alias.

Syntax

```
ImplementationDef find_impldef_by_alias (ImplRepository  receiver,
                                         Environment  *env, string  alias_name)
```

Parameters

receiver (ImplRepository)
A pointer to the **ImplRepository** object.

env (Environment *)
A pointer to the **Environment** structure for the method caller.

alias_name (string)
User-friendly name used to identify the implementation.

Returns

rc (ImplementationDef)
A copy of the desired **ImplementationDef** object is returned. *Ownership* of the object is transferred to the caller.

An exception is returned if there was an error reading the Implementation Repository.

Remarks

Finds and returns the **ImplementationDef** object whose alias is *alias_name.*

Original Class

ImplRepository

Example Code

```
#include  <somd.h>

Environment ev;
ImplementationDef impldef;
...
impldef =
   _find_impldef_by_alias(SOMD_ImplRepObject,&ev,"stack Server");
_delete_impldef(SOMD_ImplRepObject,&ev,__get_impl_id(impl def,&ev));
```

find_impldef_by_class

This method returns a sequence of implementation definitions for servers that are associated with a specified class.

Syntax

```
sequence<ImplementationDef> find_impldef_by_class
                                (ImplRepository  receiver,
                                Environment  *env, string  classname)
```

Parameters

receiver (ImplRepository)
A pointer to the **ImplRepository** object.

env (Environment *)
A pointer to the **Environment** structure for the method caller.

classname (string)
A **string** whose value is the class name of interest.

Returns

rc (sequence<ImplementationDef>)
Copiess of all **ImplementationDef** objects are returned in a sequence. *Ownership* of the sequence structure, the object array buffer, and the objects themselves is transfered to the caller.

An exception is returned if there was an error reading the Implementation Repository.

Remarks

Returns a sequence of **ImplementationDefs** for those servers that have registered an association with a specified class. Typically, a server will be associated with the classes it knows how to implement by registering its known classes via the **add_class_to_impldef** method.

Original Class

ImplRepository

Example Code

```
#include  <somd.h>

Environment ev;
sequence(ImplementationDef) impldefs;
...
impldef =
   _find_impldef_by_class(SOMD_ImplRepObject,&ev,"Stack");
```

remove_class_from_all

This method removes the association of a particular class from all servers.

Syntax

```
void remove_class_from_all  (ImplRepository  receiver,
                             Environment  *env, string  className)
```

Parameters

receiver (ImplRepository)
A pointer to an object of class **ImplRepository.**

env (Environment *)
A pointer to the **Environment** structure for the calling method.

className (string)
A string whose value is the class name of interest.

Returns

rc (void)

Remarks

The **remove_class_from_all** method removes the *className* from all of the **ImplementationDefs**.

Original Class

ImplRepository

Example Code

```
#include <somd.h>

Environment ev;
...
remove_class_from_all(SOMD_ImplRepObject, &ev, "Stack");
```

remove_class_from_impldef

This method removes the association of a particular class with a server.

Syntax

```
void remove_class_from_impldef  (ImplRepository  receiver,
                                 Environment  *env, ImplId  implid,
                                 string  classname)
```

Parameters

receiver (ImplRepository)
A pointer to the **ImplRepository** object.

env (Environment *)
A pointer to the **Environment** structure for the method caller.

implid (ImplId)
A pointer to an **ImplRepository** object.

classname (string)
A **string** whose value is the class name of interest.

Returns

rc (void)
An exception is returned if there was an error updating the Implementation Repository.

Remarks

Removes the specified class name from the set of class names associated with the server implementation identified by *implid*.

Original Class

ImplRepository

Example Code

```
#include  <somd.h>

Environment ev;
SOMDServer server;
ImplementationDef impldef;
ImplId implid;
...
server = _find_server_by_name(SOMD_ObjectMgr,&ev,"stackServer");
impldef = _get_implementation(server,&ev);
implid = __get_impl_id(impldef,&ev);
_remove_class_from_impldef(SOMD_ImplRepObject,
                           &ev,implid,"Queue");
```

update_impldef

This method updates an implementation definition in the Implementation Repository.

Syntax

```
void update_impldef  (ImplRepository  receiver, Environment  *env,
                      ImplementationDef  impldef)
```

Parameters

receiver (ImplRepository)
A pointer to the **ImplRepository** object.

env (Environment *)
A pointer to the **Environment** structure for the method caller.

impldef (ImplementationDef)
A pointer to an **ImplementationDef** object, whose values are to be saved in the Implementation Repository.

Returns

rc (void)
An exception is returned if there was an error updating the Implementation Repository.

Remarks

Replaces the state of the specified **ImplementationDef** object in the Implementation Repository. The ID of the *impldef* determines which object gets updated in the Implementation Repository.

Original Class

ImplRepository

Example Code

```
#include  _somd.h>
Environment ev;
SOMDObject objref;
ImplementationDef impldef;
...
impldef = _get_implementation(objref,&ev);
__set_impl_program(impldef,&ev,"/u/joe/bin/myserver");
_update_impldef(SOMD_ImplRepObject,&ev,impldef);
```

NVList

File stem: nvlist

Base

SOMObject

Metaclass

SOMClass

Ancestor Classes

SOMObject

Description

The type **NamedValue** is a standard datatype defined in CORBA (see the CORBA 1.1 page 106). It can be used either as a parameter type or as a mechanism for describing arguments to a request. The **NVList** class implements the **NVList** object used for constructing lists composed of **NamedValues**. **NVLists** can be used to describe arguments passed to request operations or to pass lists of property names and values to context object routines. Addition information about **NVList** is contained in Chapter 6 of the CORBA 1.1 specification.

New methods

The following list shows all the NVList methods.

- add_item
- free
- free_memory
- get_count
- get_item*
- set_item*

(* These methods were added by DSOM to supplement the published CORBA 1.1 methods.)

Overridden methods

The following list shows all the methods overridden by the NVList class. These methods are overridden in order to modify the behavior defined by an ancestor class.

- somInit

add_item

This method adds an item to the specified NVList.

Syntax

```
ORBStatus add_item  (NVList  receiver, Environment  *env,
                     Identifier  item_name, TypeCode  item_type,
                     void  *value, long  value_len, Flags  item_flags)
```

Parameters

receiver (NVList)
A pointer to the **NVList** object to which the item will be added.

env (Environment *)
A pointer to the **Environment** structure for the method caller.

item_name (Identifier)
The name of the item to be added.

item_type (TypeCode)
The data type of the item to be added.

value (void *)
A pointer to the value of the item to be added.

value_len (long)
The length of the item value to be added.

item_flags (Flags)
A **Flags** bitmask (unsigned long). The *item_flags* can be one of the following values to indicate parameter direction:

ARG_IN	The argument is input only.
ARG_OUT	The argument is output only.
ARG_INOUT	The argument is input/output.
IN_COPY_VALUE	An internal copy of the argument is made and used.
DEPENDENT_LIST	Indicates that a specified sublist must be freed when the parent list is freed.

add_item

Returns

rc (ORBStatus)
Returns an **ORBStatus** value representing the return code from the operation.

Remarks

The **add_item** method adds an item to the end of the specified list.

Original Class

NVList

Related Methods

Methods

- **free**
- **free_memory**
- **get_count**
- **get_item**
- **set_item**
- **create_list**

Example Code

```
#include  <somd.h>

Environment ev;
NVList plist;
ORBStatus rc;
...
rc = _create_list(SOMD_ORBObject, &ev, 0, &plist);
rc = _add_item(plist, &ev, "firstname", TC_string,"Joe", 3, 0);
rc = _add_item(plist, &ev, "lastname", TC_string,"Schmoe", 5, 0);
```

free

This method frees a specified **NVList**.

Syntax

```
ORBStatus free  (NVLIST  receiver, Environment  *env)
```

Parameters

receiver (NVLIST)
A pointer to the **NVList** object to be freed.

env (Environment *)
A pointer to the **Environment** structure for the method caller.

Returns

rc (ORBStatus)
Returns an **ORBStatus** value representing the return code from the operation.

Remarks

The **free** method frees an **NVList** object and any associated memory. It makes an implicit call to the **free_memory** method.

Original Class

NVList

Related Methods

Functions

- **ORBfree**

Methods

- **free_memory**

free

Example Code

```
#include  <somd.h>

Environment ev;
long nargs;
NVList arglist;
ORBStatus rc;
...
rc = _create_list(SOMD_ORBObject, &ev, nargs, &arglist);
...
rc=  _free(arglist,&ev);
```

free_memory

This method frees any dynamically allocated out-arg memory associated with the specified list.

Syntax

```
ORBStatus free_memory  (NVList  receiver, Environment  *env)
```

Parameters

receiver (NVList)
A pointer to the **NVList** object whose out-arg memory is to be freed.

env (Environment *)
A pointer to the **Environment** structure for the method caller.

Returns

rc (ORBStatus)
Returns an **ORBStatus** value representing the return code from the operation.

Remarks

The **free_memory** method frees any dynamically allocated out-arg memory associated with the specified list, without freeing the list object itself. This would be useful when invoking a DII request multiple times with the same **NVList.**

Original Class

NVList

Related Methods

Functions

- **ORBfree**

Methods

- **free**

free_memory

Example Code

```
#include  <somd.h>
#include  <repostry.h>
#include  <intfacdf.h>
#include  <foo.h>   /* provided by user */

/* assume following method declaration in interface Foo:
 *  long methodLong  (in long inLong,inout long inoutLong);
 * then the following code repeatedly invokes a request:
 * result = methodLong(fooObj, &ev, 100, 200);
 * using the DII.
 */

Environment ev;
NVList arglist;
NamedValue result;
long rc;
Foo fooObj;
Request reqObj;

/* See example code for "invoke" to see how the argList is built */

/* Create the Request, reqObj */
rc = _create_request(fooObj, &ev, (Context *)NULL, "methodLong",
                     arglist, &result, &reqObj, (Flags)0);

/* Repeatedly invoke the Request */
for (;;) {
   rc = _invoke(reqObj, &ev, (Flags)0);
   ...
   rc = _free_memory(arglist,&ev);  /* free out args */
}
...
```

get_count

This method returns the total number of items allocated for a list.

Syntax

```
ORBStatus get_count  (NVList  receiver, Environment  *env, long  count)
```

Parameters

receiver (NVList)
A pointer to the **NVList** object on which count is desired.

env (Environment *)
A pointer to the **Environment** structure for the method caller.

count (long)
A pointer to where the method will store the **long** integer count value.

Returns

rc (ORBStatus)
Returns an **ORBStatus** value representing the return code from the operation.

Remarks

The **get_count** method returns the total number of allocated items in the specified list.

Original Class

NVList

Related Methods

Methods

- **add_item**
- **get_item**
- **set_item**
- **create_list**

Example Code

```
#include  <somd.h>

Environment ev;
long nargs, list_size;
NVList arglist;
ORBStatus rc;
...
rc = _create_list(SOMD_ORBObject, &ev, nargs, &arglist);
...
rc = _get_count(arglist,&ev,&list_size);
```

get_item

This method returns the contents of a specified list item.

Syntax

```
ORBStatus get_item  (NVList  receiver, Environment  *env,
                     long  item_number, Identifier  item_name,
                     TypeCode  item_type, void  *value, long  value_len,
                     Flags  item_flags)
```

Parameters

receiver (NVList)
A pointer to an **NVList** object.

env (Environment *)
A pointer to the **Environment** structure for the method caller.

item_number (long)
The position (index) of the item in the list. The item_number ranges from 0 to *n*-1, where *n* is the total number of items in the list.

item_name (Identifier)
A pointer to where the name of the item should be returned.

item_type (TypeCode)
A pointer to where the data type of the item should be returned.

value (void *)
A pointer to where a pointer to the value of the item should be returned.

value_len (long)
A pointer to where the length of the item value should be returned.

item_flags (Flags)
A **Flags** bitmask (unsigned long). The *item_flags* can be one of the following values indicating parameter direction.

ARG_IN	The argument is input only.
ARG_OUT	The argument is output only.
ARG_INOUT	The argument is input/output.

get_item

IN_COPY_VALUE	Indicates a copy of the argument is contained and used by the NVList.
DEPENDENT_LIST	Indicates that a specified sublist must be freed when the parent list is freed.

Returns

rc (ORBStatus)
Returns 0 for success, or a DSOM error code for failure (often because item_number+1 exceeds the number of items in the list).

Remarks

The **get_item** method gets an item from the specified list. Items are numbered from 0 through *N*. The mode flags can be one of the following values:

The **get_item** method transfers ownership of storage allocated for the item value to the caller.

Original Class

NVList

Related Methods

Methods

- **add_item**
- **set_item**
- **create_list**

Example Code

```
#include  <somd.h>

Environment ev;
long i, nArgs;
ORBStatus rc;
Identifier name;
TypeCode typeCode;
void *value;
long len;
Flags flags;
NVList argList;
...
/* get number of args */
rc = _get_count(argList, ev, &nArgs);
for (i = 0; i
   /* get item description */
   rc =  _get_item(argList,
                   &ev,
                   i,
                   &name,
                   &typeCode,
                   &value,
                   &len,
                   &flags);
   ...
   }
```

set_item

This method sets the contents of an item in a list.

Syntax

```
ORBStatus set_item  (NVList  receiver, Enviornment  *env,
                     long  item_number, Identifier  item_name,
                     TypeCode  item_type, void  *value, long  value_len,
                     Flags  item_flags)
```

Parameters

receiver (NVList)
A pointer to an **NVList** which contains the item to be set.

env (Enviornment *)
A pointer to the **Environment** structure for the method caller.

item_number (long)
The position (index) of the item in the list. The *item_number* ranges from 0 to *n*-1, where *n* is the total number of items in the list.

item_name (Identifier)
The name of the set item.

item_type (TypeCode)
The data type of the set item.

value (void *)
A pointer to the value of the set item.

value_len (long)
The length of the set item value.

item_flags (Flags)
A **Flags** bitmask (unsigned long). The *item_flags* can be one of the following values to indicate parameter direction:

ARG_IN	The argument is input only.
ARG_OUT	The argument is output only.
ARG_INOUT	The argument is input/output.

IN_COPY_VALUE	Indicates an internal copy of the argument is made and used.
DEPENDENT_LIST	Indicates that a specified sublist must be freed when the parent list is freed.

Returns

rc (ORBStatus)
Returns 0 on successful completion or a DSOM error code upon failure (often because item_number+1 exceeds the number of items in the list).

Remarks

The **set_item** method sets the contents of an item in the list.

Original Class

NVList

Related Methods

Methods

- **get_item**
- **add_item**
- **create_list**

Example Code

```
#include  <somd.h>

Environment ev;
long i, nArgs;
ORBStatus rc;
Identifier name;
TypeCode typeCode;
void *value;
long len;
Flags flags;
NVList argList;
...
/* get number of args */
rc = _get_count(argList, ev, &nArgs);
for (i = 0; i
   /* change item description */
   rc =  _set_item(argList,
                   &ev,
                   i,
                   name,
                   typeCode,
                   value,
                   len,
                   flags);
   ...
   }
```

ObjectMgr

File stem: om

Base

SOMObject

Metaclass

SOMMSingleInstance

Ancestor Classes

SOMObject

Subclasses

SOMDObjectMgr

Description

The **ObjectMgr** class provides a uniform, universal abstraction for any sort of object manager. Object Request Brokers, persistent storage managers, and OODBMSs are examples of object managers.

This is an abstract base class, which defines the "core" interface for an object manager. It provides basic methods that:

- Create a new object of a certain class,
- Return a (persistent) ID for an object,
- Return a reference to an object associated with an ID,
- Free an object (i.e., release any local memory associated with the object without necessarily destroying the object itself), or
- Destroy an object.

Note: The **ObjectMgr** is an *abstract* class and should not be instantiated. Any subclass of **ObjectMgr** must provide implementations for all **ObjectMgr** methods. In DSOM, the class **SOMDObjectMgr** provides a DSOM-specific implementation.

New methods

The following list shows all the ObjectMgr instance methods.

- somdDestroyObject*
- somdGetIdFromObject*
- somdGetObjectFromID*
- somdNewObject*
- somdReleaseObject*

(* This class and its methods were added by DSOM to supplement the published CORBA 1.1 interfaces.)

somdDestroyObject

This method requests destruction of the target object.

Syntax

```
void somdDestroyObject  (ObjectMgr  receiver, Environment  *env,
                         SOMObject  obj)
```

Parameters

receiver (ObjectMgr)
A pointer to an **ObjectMgr** object.

env (Environment *)
A pointer to the **Environment** structure for the method caller.

obj (SOMObject)
A pointer to the object to be freed.

Returns

rc (void)

Remarks

The **somdDestroyObject** method indicates that the object manager should destroy the specified object. Storage associated with the object is freed.

In DSOM, the **SOMDObjectMgr** forwards the deletion request to the remote server, and then frees the local proxy object.

Original Class

ObjectMgr

Related Methods

Methods

- **somdReleaseObject**
- **somdCreateObject**
- **somdTargetFree**

- **release**

Example Code

```
#include  <somd.h>

Stack stk;
Environment ev;
SOMDServer server;

SOM_InitEnvironment(&ev);
SOMD_Init(&ev);
StackNewClass(0,0);
server =
  _somdFindAnyServerByClass(SOMD_ObjectMgr, &ev,"Stack");
stk = _somdCreateObj(server, &ev, "Stack", "");
...
_somdDestroyObject(SOMD_ObjectMgr, &ev, stk);
```

somdGetIdFromObject

This method returns an ID for an object managed by a specified Object Manager.

Syntax

```
string somdGetIdFromObject  (ObjectMgr  receiver, Environment  *env,
                             SOMDObject  obj)
```

Parameters

receiver (ObjectMgr)
A pointer to an **ObjectMgr** object.

env (Environment *)
A pointer to the **Environment** structure for the method caller.

obj (SOMDObject)
A pointer to the **SOMDObject** object for which an ID is needed.

Returns

rc (string)
Returns a string representing the ID of the specified object.

Remarks

The **somdGetIdFromObject** method returns the persistent ID for an object managed by the specified Object Manager. This ID is unambiguous—it always refers to the same object.

The **somdGetIdFromObject** method transfers *ownership* of storage allocated for the string to the caller. The caller should free the result using **ORBfree** function, rather than the **SOMFree** function.

Original Class

ObjectMgr

Related Methods

Methods

- **somdGetObjectFromId**

Example Code

```
#include  <somd.h>
#include  <car.h>

Environment ev;
Car car;
string somdObjectId;
/*note that "SOMDObject Identifiers" are just strings */

SOM_InitEnvironment(&ev);
SOMD_Init(&ev);

/* create a remote Car object */
car = _somdNewObject(SOMD_ObjectMgr, &ev, "Car", "");

/* save the reference to the object */
somdObjectId = _somdGetIdFromObject(SOMD_ObjectMgr, &ev, car);
FileWrite("/u/joe/mycar", somdObjectId);
...
```

somdGetObjectFromId

This method finds and activates an object implemented by a specified object manager, given its ID.

Syntax

```
SOMObject somdGetObjectFromId  (ObjectMgr  receiver,
                                Environment  *env, string  id)
```

Parameters

receiver (ObjectMgr)
A pointer to an **ObjectMgr** object.

env (Environment *)
A pointer to the **Environment** structure for the method caller.

id (string)
A string representing an object ID.

Returns

rc (SOMObject)
Returns a pointer to the object with the specified ID.

Remarks

The **somdGetObjectFromId** method finds and activates an object implemented by this object manager, given its ID.

The **somdGetObjectFromId** method transfers *ownership* to the caller.

Original Class

ObjectMgr

Related Methods

Methods

- **somdGetIdFromObject**

Example Code

```
#include  <somd.h>
#include  <car.h>
Environment ev;
Car car;
string somdObjectId;
...
/* restore proxy from its string form */
FileRead("/u/joe/mycar", &somdObjectId);
car = _somdGetObjectFromId(SOMD_ObjectMgr, &ev, somdObjectId);
...
```

somdNewObject

This method returns a new object of the named class.

Syntax

```
SOMObject somdNewObject  (ObjectMgr  receiver, Environment  *env,
                          Identifier  objclass, string  hints)
```

Parameters

receiver (ObjectMgr)
A pointer to an **ObjectMgr** object.

env (Environment *)
A pointer to the **Environment** structure for the method caller.

objclass (Identifier)
An **Identifier** representing the type of the new object.

hints (string)
A **string** which may optionally be used to specify special creation options.

Returns

rc (SOMObject)
Returns a **SOMObject**. *Ownership* of the new object is transferred to the caller.

Remarks

The **somdNewObject** method returns a new object of the class specified by *objclass*. The *hints* field is currently ignored by the **SOMDObjectMgr**. (This field is designed so that application-specific creation options can be supplied in a later SOMobjects release.)

In DSOM, the **SOMDObjectMgr** selects a random server which has advertised knowledge of the desired class *objclass*, and forwards the creation request to that server. If multiple capable servers are registered in the Implementation Repository, subsequent calls will be routed to different servers (starting them automatically if necessary) in order to distribute the load across several servers where possible.

somdNewObject

Original Class

ObjectMgr

Related Methods

Methods

- **somdDestroyObject**
- **somdReleaseObject**

Example Code

```
#include  <somd.h>
#include  <stack.h>   /* provided by user */

Stack stk;
Environment ev;
SOMDServer server;

SOM_InitEnvironment(&ev);
SOMD_Init(&&ev);
StackNewClass(0,0);
stk = _somdNewObject(SOMD_ObjectMgr, &ev, "Stack", "");
...
_somdDestroyObject(SOMD_ObjectMgr, &ev, stk);
```

somdReleaseObject

This method indicates that the client has finished using the object.

Syntax

```
void somdReleaseObject  (ObjectMgr  receiver, Environment  *env,
                         SOMObject  obj)
```

Parameters

receiver (ObjectMgr)
A pointer to an **ObjectMgr** object.

env (Environment *)
A pointer to the **Environment** structure for the method caller.

obj (SOMObject)
A pointer to the object to be released.

Returns

rc (void)

Remarks

The **somdReleaseObject** method indicates that the client has finished using the specified object. This allows the object manager to free the bookkeeping information associated with the object, if any. The object may also be passivated, but it is not destroyed.

In DSOM, **somdReleaseObject** causes the client's proxy for the target object of interest to be freed; the target object is not freed.

Original Class

ObjectMgr

Related Methods

Methods

- **somdDestroyObject**

- **somdNewObject**
- **somdTargetFree**
- **release**

Example Code

```
#include  <somd.h>
#include  <car.h>

Environment ev;
Car car;
string somdObjectId;
...
/* restore proxy from its string form */
FileRead("/u/joe/mycar", &somdObjectId);
car = _somdGetObjectFromId(SOMD_ObjectMgr, &ev, somdObjectId);
...
_somdReleaseObject(SOMD_ObjectMgr, &ev, car);
```

ORB

File stem: orb

Base

SOMObject

Metaclass

SOMMSingleInstance

Ancestor Classes

SOMObject

Description

The **ORB** class implements the CORBA ORB object described in Chapter 8 of the CORBA 1.1 specification. The **ORB** class defines operations for converting object references to strings and converting strings to object references. The **ORB** also defines operations used by the Dynamic Invocation Interface for creating lists (NVLists) and determining the default context.

New methods

The following list shows all the ORB methods.

- create_list
- create_operation_list
- get_default_context
- object_to_string
- string_to_object

create_list

This method creates an **NVList** of the specified size.

Syntax

```
ORBStatus create_list  (ORB  receiver, Environment  *env, long  count,
                        NVList  new_list)
```

Parameters

receiver (ORB)
A pointer to the **ORB** object.

env (Environment *)
A pointer to the **Environment** structure for the method caller.

count (long)
An integer representing the number of elements to allocate for the list.

new_list (NVList)
A pointer to the address where the method will store a pointer to the allocated **NVList** object.

Returns

rc (ORBStatus)
Returns an **ORBStatus** value representing the return code of the operation.

Remarks

Creates an **NVList** list of the specified size, typically for use in **Request**s.

Ownership of the allocated *new_list* is transferred to the caller.

Original Class

ORB

Related Methods

Methods

- **create_operation_list**

Example Code

```
#include  <somd.h>

Environment ev;
long nargs = 5;
NVList arglist;
ORBStatus rc;
...
rc = _create_list(SOMD_ORBObject, &ev, nargs, &arglist);
```

create_operation_list

This method creates an **NVList** initialized with the argument descriptions for a given operation.

Syntax

```
ORBStatus create_operation_list (ORB  receiver, Environment  *env,
                                 OperationDef  oper, NVList  new_list)
```

Parameters

receiver (ORB)
A pointer to the **ORB** object.

env (Environment *)
A pointer to the **Environment** structure for the method caller.

oper (OperationDef)
A pointer to the **OperationDef** object representing the operation for which the NVList is to be initialized.

new_list (NVList)
A pointer to where the method will store a pointer to the resulting argument list.

Returns

rc (ORBStatus)
Returns an **ORBStatus** value representing the return code of the operation.

Ownership of the allocated *new_list* is transferred to the caller.

Remarks

Creates an **NVList** list for the specified operation, for use in **Request**s invoking that operation.

Original Class

ORB

Related Methods

Methods

- **create_list**

Example Code

```
#include  <somd.h>

Environment ev;
OperationDef opdef;
NVList arglist;
long rc;

/* Get the OperationDef from the Interface Repository. */
opdef = _lookup_id(SOM_InterfaceRepository,
                   &ev, "Foo::methodLong");
/* Create a NamedValue list for the operation. */
rc= _create_operation_list(SOMD_ORBObject, &ev, opdef, &arglist);
```

get_default_context

This method returns the default process **Context** object.

Syntax

```
ORBStatus get_default_context (ORB  receiver, Environment  *env,
                               Context  ctx)
```

Parameters

receiver (ORB)
A pointer to the **ORB** object.

env (Environment *)
A pointer to the **Environment** structure for the method caller.

ctx (Context)
A pointer to where the method will store a pointer to the returned **Context** object.

Returns

rc (ORBStatus)
Returns an **ORBStatus** return code: 0 indicates success, while a non-zero value is a DSOM error code (see Chapter 6 of the *SOM Programming Guide*).

Remarks

The **get_default_context** method gets the default process **Context** object.

Ownership of the allocated **Context** object is transferred to the caller.

Original Class

ORB

Example Code

```
#include  <somd.h>

Environment ev;
Context cxt;
long rc;
...
rc = _get_default_context(SOMD_ORBObject, &ev, &cxt);
```

object_to_string

This method converts an object reference to an external form (string) which can be stored outside the ORB.

Syntax

```
string object_to_string  (ORB  receiver, Environment  *env,
                          SOMDObject  obj)
```

Parameters

receiver (ORB)
A pointer to the **ORB** object.

env (Environment *)
A pointer to the **Environment** structure for the method caller.

obj (SOMDObject)
A pointer to a **SOMDObject** object representing the reference to be converted.

Returns

rc (string)
Returns a string representing the external (string) form of the referenced object.

Remarks

The **object_to_string** method converts the object reference to a form (string) which can be stored externally.

Ownership of allocated memory is transferred to the caller. The caller should free the result using the **ORBfree** function, rather than the **SOMFree** function.

Original Class

ORB

Related Methods

Methods

- **string_to_object**

Example Code

```
#include  <somd.h>
#include  <car.h>

Environment ev;
Car car;
string objrefstr;

SOM_InitEnvironment(&ev);
SOMD_Init(&ev);

/* create a remote Car object */
car = _somdNewObject(SOMD_ObjectMgr, &ev, "Car", "");

/* save the reference to the object */
objrefstr = _object_to_string(SOMD_ORBObject, &ev, car);
FileWrite("/u/joe/mycar", objrefstr);
```

string_to_object

This method converts an externalized (string) form of an object reference into an object reference.

Syntax

```
SOMDObject string_to_object (ORB  receiver, Environment  *env,
                             string  str)
```

Parameters

receiver (ORB)
A pointer to the **ORB** object.

env (Environment *)
A pointer to the **Environment** structure for the method caller.

str (string)
A pointer to a character string representing the externalized form of the object reference.

Returns

rc (SOMDObject)
Returns a **SOMDObject** object.

Remarks

The **string_to_object** method converts the externalized (string) form of an object reference into an object reference.

Original Class

ORB

Related Methods

Methods

- **object_to_string**

Example Code

```
#include  <somd.h>
#include  <car.h>

Environment ev;
Car car;
string objrefstr;
...
/* restore proxy from its string form */
FileRead("/u/joe/mycar", &objrefstr);
car = _string_to_object(SOMD_ORBObject, &ev, objrefstr);
```

File stem: principl

Base

SOMObject

Metaclass

SOMClass

Ancestor Classes

SOMObject

Description

The **Principal** class defines attributes which identify the user ID and host name of the originator of a specific request. This information is typically used for access control.

A **Principal** object is returned by the **get_principal** method of the SOM Object Adapter. The parameters of the **get_principal** method identify the environment and target object associated with a particular request—the **SOMOA** uses this information to create a **Principal** object which identifies the caller.

Note: Details of the **Principal** object are not currently defined in the CORBA 1.1 specification; the attributes which have been defined are required by DSOM.

Attributes

Listed below is each available attribute, with its corresponding type in parentheses, followed by a description of its purpose:

userName (string)
Identifies the name of the user associated with the request invocation. (Currently, this value is obtained from the USER environment variable in the process which invoked the request.)

hostName (string)
Identifies the name of the host from where the request originated. (Currently, this value is obtained from the HOSTNAME environment variable in the process which invoked the request.)

Request

File stem: request

Base

SOMObject

Metaclass

SOMClass

Ancestor Classes

SOMObject

Description

The **Request** class implements the CORBA Request object described in section 6.2 on page 108 of CORBA 1.1. The **Request** object is used by the dynamic invocation interface to dynamically create and issue a request to a remote object. **Request** objects are created by the **create_request** method in **SOMDObject**.

New methods

The following list shows all the Request methods.

- add_arg
- destroy*
- get_response
- invoke
- send

(* The **destroy** method was defined as **delete** in CORBA 1.1, which conflicts with the **delete** operator in C++ ** However, there is a **Request_delete** macro defined for CORBA compatibility.)

Overridden methods

The following list shows all the methods overridden by the Request class. These methods are overridden in order to modify the behavior defined by an ancestor class.

- somInit
- somUninit

add_arg

This method incrementally adds an argument to a **Request** object.

Syntax

```
ORBStatus add_arg  (Request  receiver, Environment  *env,
                    Identifier  name, TypeCode  arg_type, void  *value,
                    long  len, Flags  arg_flags)
```

Parameters

receiver (Request)
A pointer to a **Request** object.

env (Environment *)
A pointer to the **Environment** structure for the method caller.

name (Identifier)
An identifier representing the name of the argument to be added.

arg_type (TypeCode)
The typecode for the argument to be added.

value (void *)
A pointer to the argument value to be added.

len (long)
The length of the argument.

arg_flags (Flags)
A **Flags** bitmask (unsigned long). The *arg_flags* parameter may take one of the following values to indicate parameter direction:

ARG_IN	The argument is input only.
ARG_OUT	The argument is output only.
INOUT	The argument is input/output.
IN_COPY_VALUE	An internal copy of the argument is to be made and used.
DEPENDENT_LIST	Indicates that a specified sublist must be freed when the parent list is freed.

Returns

rc (ORBStatus)
Returns an **ORBStatus** value representing the return code of the operation.

Remarks

The **add_arg** method incrementally adds an argument to a **Request** object. The **Request** object must have been created using the **create_request** method with an empty argument list.

Original Class

Request

Example Code

add_arg

```
#include  <somd.h>
#include  <repostry.h>
#include  <intfacdf.h>
#include  <foo.h>   /* provided by user */

/* assume following method declaration in interface Foo:
 *      long methodLong  (in long inLong,inout long inoutLong);
 * then the following code builds a request to execute the call:
 *      result = methodLong(fooObj, &ev, 100,200);
 *using the DII.
 */

Environment ev;
OperationDef opdef;
Description desc;
OperationDescription *opdesc;
long rc;
long value1 = 100;
long value2 = 200;
Foo fooObj;
Request reqObj;
NamedValue result;

/* Get the OperationDef from the Interface Repository. */
opdef = _lookup_id(SOM_InterfaceRepository,
&ev, "Foo::methodLong");

/* Get the operation description structure. */
desc = _describe(opdef, &ev);
opdesc = (OperationDescription *) desc.value._value;

/* Fill in the TypeCode field for result. */
result.argument._type = opdesc->result;

/* Create the Request, reqObj */
rc = _create_request(fooObj, &ev, (Context* )NULL, "methodLong",
                     (NVList *)NULL, &result, &reqObj, (Flags)0);

/* Add arg1 info onto the request */
_add_arg(reqObj, &ev,
          "inLong", TC_long, &value1, sizeof(long), (Flags)0);
/* Add arg2 info onto the request */
_add_arg(reqObj, &ev,
          "inoutLong", TC_long, &value2, sizeof(long), (Flags)0);
```

destroy

This method deletes the memory allocated by the ORB for a **Request** object.

Syntax

```
ORBStatus destroy  (Request  receiver, Environment  *env)
```

Parameters

receiver (Request)
A pointer to a **Request** object.

env (Environment *)
A pointer to the **Environment** structure for the method caller.

Returns

rc (ORBStatus)
Returns an **ORBStatus** value representing the return code of the operation.

Remarks

The **destroy** method deletes the **Request** object and all associated memory.

Note: This method is called "delete" in the CORBA 1.1 specification. However, the word "delete" is a reserved operator in C++, so the name "destroy" was chosen as an alternative. For CORBA compatibility, a macro defining **Request_delete** as an alias for **destroy** has been included in the C header files.

Original Class

Request

Related Methods

Methods

- **invoke**
- **send**
- **get_response**

destroy

Example Code

```
#include  <somd.h>
#include  <repostry.h>
#include  <intfacdf.h>
#include  <foo.h>   /* provided by user */

/* assume following method declaration in interface Foo:
 *      long methodLong  (in long inLong,inout long inoutLong);
 * then the following code  sends a request to execute the call:
 *      result = methodLong(fooObj, &ev, 100,200);
 * using the DII without waiting for the result. Then, later,
 * waits for and then uses the result.
 */
Environment ev;
NVList arglist;
long rc;
Foo fooObj;
Request reqObj;
NamedValue result;

/* see the Example code for invoke to see how the request
 * is built
 */

/* Create the Request, reqObj */
rc = _create_request(fooObj, &ev, (Context *)NULL, "methodLong",
                     arglist, &result, &reqObj, (Flags)0);

/* Finally, send the request */
rc = _send(reqObj, &ev, (Flags)0);

/* do some work, i.e. don't wait for the result */

/* wait here for the result of the request */
rc = _get_response(reqObj, &ev, (Flags)0);

/* use the result */
if (result->argument._value == 9600) {...}

/* throw away the reqObj */
_destroy(reqObj, &ev);
```

get_response

This method determines whether an asynchronous **Request** has completed.

Syntax

```
ORBStatus get_response (Request  receiver, Environment  *env,
                        Flags  response_flags)
```

Parameters

receiver (Request)
A pointer to a **Request** object.

env (Environment *)
A pointer to the **Environment** structure for the method caller.

response_flags (Flags)
A **Flags** bitmask (unsigned long) containing control information for the get_response method. The *response_flags* argument may have the following value:

RESP_NO_WAIT Indicates the caller does not want to wait for a response.

Returns

rc (ORBStatus)
Returns an **ORBStatus** value representing the return code of the operation.

Remarks

The **get_response** method determines whether the asynchronous **Request** has completed.

Original Class

Request

Related Methods

Methods

- **invoke**
- **send**

get_response

Macros

- **Request_delete**

Example Code

```
#include  <somd.h>
#include  <repostry.h>
#include  <intfacdf.h>
#include  <foo.h>   /* provided by user */

/* assume following method declaration in interface Foo:
 *      long methodLong  (in long inLong,inout long inoutLong);
 * then the following code  sends a request to execute the call:
 *      result = methodLong(fooObj, &ev, 100,200);
 * using the DII without waiting for the result. Then, later,
 * waits for and then uses the result.
 */

Environment ev;
NVList arglist;
long rc;
Foo fooObj;
Request reqObj;
NamedValue result;

/* see the Example code for invoke to see how the request
 * is built
 */

/* Create the Request, reqObj */
rc = _create_request(fooObj, &ev, (Context *)NULL, "methodLong",
                     arglist, &result, &reqObj, (Flags)0);

/* Finally, send the request */
rc = _send(reqObj, &ev, (Flags)0);

/* do some work, i.e. don't wait for the result */

/* wait here for the result of the request */
rc = _get_response(reqObj, &ev, (Flags)0);

/* use the result */
if (result->argument._value == 9600) {...}
```

invoke

This method invokes a **Request** synchronously, waiting for the response.

Syntax

```
ORBStatus invoke  (Request  receiver, Environment  *env,
                   Flags  invoke_flags)
```

Parameters

receiver (Request)
A pointer to a **Request** object.

env (Environment *)
A pointer to the **Environment** structure for the method caller.

invoke_flags (Flags)
A **Flags** bitmask (unsigned long) representing control information for the **invoke** method. There are currently no flags defined for the **invoke** method.

Returns

rc (ORBStatus)
Returns an **ORBStatus** value representing the return code of the operation.

Remarks

The **invoke** method sends a **Request** synchronously, waiting for the response.

Original Class

Request

invoke

Related Methods

Methods

- **send**
- **get_response**

Macros

- **Request_delete**

Example Code

```
#include  <somd.h>
#include  <repostry.h>
#include  <intfacdf.h>
#include  <foo.h>   /* provided by user */

/* assume following method declaration in interface Foo:
 *      long methodLong  (in long inLong,inout long inoutLong);
 * then the following code builds and then  invokes
 * a request to execute the call:
 *      result = methodLong(fooObj, &ev, 100,200);
 * using the DII.
 */

Environment ev;
OperationDef opdef;
Description desc;
OperationDescription *opdesc;
NVList arglist;
long rc;
long value1 = 100;
long value2 = 200;
Foo fooObj;
Request reqObj;
NamedValue result;
Identifier name;
TypeCode tc;
void *dummy;
long dummylen;
Flags flags;

/* Get the OperationDef from the Interface Repository. */
opdef = _lookup_id(SOM_InterfaceRepository,
                   &ev, "Foo::methodLong");

/* Create a NamedValue list for the operation. */
rc= _create_operation_list(SOMD_ORBObject, &ev, opdef, &arglist);

/* Insert arg1 info into arglist */
_get_item(arglist, &ev,
          0, &name, &tc, &dummy, &dummylen, &flags);
_set_item(arglist,&ev,0, name, tc, &value1, sizeof(long), flags);

/* Insert arg2 info into arglist */
_get_item(arglist, &ev,
          1, &name, &tc, &dummy, &dummylen, &flags);
_set_item(arglist,&ev,1, name, tc, &value2, sizeof(long), flags);
```

invoke

```
/* Get the operation description structure. */
desc = _describe(opdef, &ev);
opdesc = (OperationDescription *) desc.value._value;
/* Fill in the TypeCode field for result. */
result.argument._type = opdesc->result;

/* Create the Request, reqObj */
rc = _create_request(fooObj, &ev, (Context *)NULL, "methodLong",
                     arglist, &result, &reqObj, (Flags)0);

/* Finally, invoke the request */
rc = _invoke(reqObj, &ev, (Flags)0);

/* Print results */
printf("result: %d, value2: %d\n",
       *(long*)(result.argument._value),
       value2);
```

send

This method invokes a **Request** asynchronously.

Syntax

```
ORBStatus send  (Request  receiver, Environment  *env,
                 Flags  invoke_flags)
```

Parameters

receiver (Request)
A pointer to a **Request** object.

env (Environment *)
A pointer to the **Environment** structure for the method caller.

invoke_flags (Flags)
A **Flags** bitmask (unsigned long) containing **send** method control information. The argument *invoke_flags* can have the following value.

INV_NO_RESPONSE	Indicates that the invoker does not intend to wait for a response, nor does it expect any of the output arguments (**inout** or **out**) to be updated.

Returns

rc (ORBStatus)
Returns an **ORBStatus** value representing the return code from the operation.

Remarks

The **send** method invokes the **Request** asynchronously. The response must eventually be checked by invoking either the **get_response** method or the **get_next_response** function.

Original Class

Request

send

Related Methods

Methods

- **invoke**
- **get_response**

Macros

- **Request_delete**

Example Code

```
#include  <somd.h>
#include  <repostry.h>
#include  <intfacdf.h>
#include  <foo.h>   /* provided by user */

/* assume following method declaration in interface Foo:
 *      long methodLong (in long inLong,inout long inoutLong);
 * then the following code sends
 *      a request to execute the call:
 * result = methodLong(fooObj, &ev, 100,200);
 * using the DII.
 */

Environment ev;
NVList arglist;
long rc;
Foo fooObj;
Request reqObj;
NamedValue result;

/* see the Example code for invoke to see how the request
 * is built
 */

/* Create the Request, reqObj */
rc = _create_request(fooObj, &ev, (Context *)NULL,"methodLong",
                                   arglist, &result, &reqObj, (Flags)0);

/* Finally, send the request */
rc = _send(reqObj, &ev, (Flags)0);
```

SOMDClientProxy

File stem: somdcprx

Base

SOMDObject

Metaclass

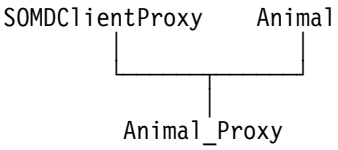
SOMClass

Ancestor Classes

SOMObject
    SOMDObject

Description

The **SOMDClientProxy** class implements DSOM proxy objects in Clients. **SOMDClientProxy** overrides the usual **somDispatch** methods with versions that build a DSOM **Request** for remote invocation and dispatch it to the remote object. It is intended that the implementation of this "generic" proxy class will be used to derive specific proxy classes via multiple inheritance. The remote dispatch method is inherited from this client proxy class, while the desired interface and language bindings are inherited from the target class (but not the implementation).



New methods

The following list shows all the SOMDClientProxy methods.

- somdProxyFree*
- somdProxyGetClass*
- somdProxyGetClassName*
- somdTargetFree*
- somdTargetGetClass*
- somdTargetGetClassName*
- somdReleaseResources*

(***** This class and its methods were added by DSOM to supplement the published CORBA'1.1 interfaces.)

Overridden methods

The following list shows all the methods overridden by the SOMDClientProxy class. These methods are overridden in order to modify the behavior defined by an ancestor class.

- create_request
- create_request_args
- is_proxy
- release
- somDispatch
- somDispatchA
- somDispatchD
- somDispatchL
- somDispatchV
- somFree
- somGetClass
- somGetClassName
- somInit
- somUninit

somdProxyFree

This method executes **somFree** on the local proxy object.

Syntax

```
void somdProxyFree  (SOMDClientProxy  receiver, Environment  *env)
```

Parameters

receiver (SOMDClientProxy)
A pointer to the **SOMDClientProxy** object.

env (Environment *)
A pointer to the **Environment** structure for the method caller.

Returns

rc (void)

Remarks

The **somdProxyFree** method executes the **somFree** method call on the local proxy object. This method has been provided when the application program wants to be explicit about freeing the proxy object vs. the target object.

Original Class

SOMDClientProxy

Related Methods

Methods

- **release**
- **somdReleaseObject**

Example Code

```
#include  <somd.h>
#include  <car.h>

Environment ev;
Car car;
string somdObjectId;
...
/* restore proxy from its string form */
FileRead("/u/joe/mycar", &somdObjectId);
car = _somdGetObjectFromId(SOMD_ObjectMgr, &ev, somdObjectId);
...
_somdProxyFree(car, &ev);
```

somdProxyGetClass

This method returns the class object for the local proxy object.

Syntax

```
SOMClass somdProxyGetClass  (SOMDClientProxy  receiver,
                             Environment  *env)
```

Parameters

receiver (SOMDClientProxy)
A pointer to the **SOMDClientProxy** object.

env (Environment *)
A pointer to the **Environment** structure for the method caller.

Returns

rc (SOMClass)
Returns a pointer to the class object for the local proxy object.

Remarks

The **somdProxyGetClass** method executes the **somGetClass** method call on the local proxy object and returns a pointer to the proxy's class object. This method has been provided when the application program wants to be explicit about getting the class object for the proxy objectvs. the target object.

Original Class

SOMDClientProxy

Example Code

```
#include  <somd.h>
#include  <car.h>

Environment ev;
Car car;
SOMClass carProxyClass;
string somdObjectId;
...
/* restore proxy from its string form */
FileRead("/u/joe/mycar", &somdObjectId);
car = _somdGetObjectFromId(SOMD_ObjectMgr, &ev, somdObjectId);
...
carProxyClass = _somdProxyGetClass(car, &ev);
```

somdProxyGetClassName

This method returns the class name for the local proxy object.

Syntax

```
string somdProxyGetClassName  (SOMDClientProxy  receiver,
                               Environment  *env)
```

Parameters

receiver (SOMDClientProxy)
A pointer to the **SOMDClientProxy** object for the desired remote target object.

env (Environment *)
A pointer to the **Environment** structure for the method caller.

Returns

rc (string)
Returns a string containing the class name of the local proxy object.

Remarks

The **somdProxyGetClassName** method executes the **somGetClassName** method call on the local proxy object and returns the proxy's class name. This method has been provided when the application program wants to be explicit about getting the class name of the proxy object vs. the target object.

Original Class

SOMDClientProxy

somdProxyGetClassName

Example Code

```
#include  <somd.h>
#include

Environment ev;
Car car;
string carProxyClassName;
string somdObjectId;
...
/* restore proxy from its string form */
FileRead("/u/joe/mycar", &somdObjectId);
car = _somdGetObjectFromId(SOMD_ObjectMgr, &ev, somdObjectId);
...
carProxyClassName = _somdProxyGetClassName(car, &ev);
```

somdReleaseResources

This method instructs a proxy object to release any memory it is holding as a result of a remote method invocation in which a parameter or result was designated as "object-owned".

Syntax

```
void somdReleaseResources  (SOMDClientProxy  receiver,
                            Environment  *env)
```

Parameters

receiver (SOMDClientProxy)
A pointer to the **SOMDClientProxy** object to release resources.

env (Environment *)
A pointer to the **Environment** structure for the method call.

Returns

rc (void)

Remarks

The **somdReleaseResources** method instructs a proxy object to release any memory it is holding as a result of a remote method invocation in which a parameter or result was designated as "object-owned".

When a DSOM client program makes a remote method invocation, via a proxy, and the method being invoked has an object-owned parameter or return result, the client-side memory associated with the parameter/result will be owned by the caller's proxy, and the server-side memory will be owned by the remote object. The memory owned by the caller's proxy will be freed when the proxy is released by the client program. (The time at which the server-side memory will be freed depends on the implementation of the remote object.)

A DSOM client can also instruct a proxy object to free all memory that it owns on behalf of the client without releasing the proxy (assuming that the client program is finished using the object-owned memory), by invoking the **somdReleaseResources**

method on the proxy object. Calling **somdReleaseResources** can prevent unused memory from accumulating in a proxy.

For example, consider a client program repeatedly invoking a remote method "get_string", which returns a string that is designated (in SOM IDL) as "object-owned". The proxy on which the method is invoked will store the memory associated with all of the returned strings, even if the strings are not unique, until the proxy is released. If the client program only uses the last result returned from "get_string", then unused memory accumulates in the proxy. The client program can prevent this by invoking **somdReleaseResources** on the proxy object periodically (for example, each time it finishes using the result of the last "get_string" call).

Original Class

SOMDClientProxy

Related Methods

Methods

- **release**

Example Code

```
string mystring;
...
/* remote invocation of get_string on proxy x,
 * where method get_string has the SOM IDL modifier
 * "object_owns_result".
 */
mystring = X_get_string(x, ev);

/* ... use mystring ... */

/* when finished using mystring, instruct the
 * proxy that it can free it.
 */
_somdReleaseResources(x, ev);
```

somdTargetFree

This method forwards the **somFree** method call to the remote target object.

Syntax

```
void somdTargetFree  (SOMDClientProxy  receiver, Environment  *env)
```

Parameters

receiver (SOMDClientProxy)
A pointer to the **SOMDClientProxy** object for the desired remote target object.

env (Environment *)
A pointer to the **Environment** structure for the method caller.

Returns

rc (void)

Remarks

The **somdTargetFree** method forwards the **somFree** method call to the remote target object. This method has been provided when the application program wants to be explicit about freeing the remote target object vs. the proxy object.

Original Class

SOMDClientProxy

Related Methods

Methods

- **release**
- **somdDestroyObject**

Example Code

```
#include  <somd.h>
#include  <car.h>

Environment ev;
Car car;
string somdObjectId;
...
/* restore proxy from its string form */
FileRead("/u/joe/mycar", &somdObjectId);
car = _somdGetObjectFromId(SOMD_ObjectMgr, &ev, somdObjectId);
...
_somdTargetFree(car, &ev);
```

somdTargetGetClass

This method returns (a proxy for) the class object for the remote target object.

Syntax

```
SOMClass somdTargetGetClass  (SOMDClientProxy  receiver,
                              Environment  *env)
```

Parameters

receiver (SOMDClientProxy)
A pointer to the **SOMDClientProxy** object for the desired remote target object.

env (Environment *)
A pointer to the **Environment** structure for the method caller.

Returns

rc (SOMClass)
Returns a pointer to the class object for the remote target object.

Remarks

The **somdTargetGetClass** method forwards the **somGetClass** method call to the remote target object and returns a pointer to the class object for that object. This method has been provided when the application program wants to be explicit about getting the class object for the remote target object vs. the local proxy.

Original Class

SOMDClientProxy

Related Methods

Methods

- **somdProxyGetClass**

Example Code

```
#include  <somd.h>
#include  <car.h>

Environment ev;
Car car;
SOMClass carClass;
string somdObjectId;
...
/* restore proxy from its string form */
FileRead("/u/joe/mycar", &somdObjectId);
car = _somdGetObjectFromId(SOMD_ObjectMgr, &ev, somdObjectId);
...
carClass = _somdTargetGetClass(car, &ev);
```

somdTargetGetClassName

This method returns the class name for the remote target object.

Syntax

```
string somdTargetGetClassName  (SOMDClientProxy  receiver,
                                Environment  *env)
```

Parameters

receiver (SOMDClientProxy)
A pointer to the **SOMDClientProxy** object for the desired remote target object.

env (Environment *)
A pointer to the **Environment** structure for the method caller.

Returns

rc (string)
Returns a string containing the class name of the remote target object.

Remarks

The **somdTargetGetClassName** method forwards the **somGetClassName** method call to the remote target object and returns the class name for that object. This method has been provided when the application program wants to be explicit about getting the class name of the remote target object vs. the proxy object.

Original Class

SOMDClientProxy

Related Methods

Methods

- **somdProxyGetClassName**

somdTargetGetClassName

Example Code

```
#include  <somd.h>
#include  <car.h>

Environment ev;
Car car;
string carClassName;
string somdObjectId;
...
/* restore proxy from its string form */
FileRead("/u/joe/mycar", &somdObjectId);
car = _somdGetObjectFromId(SOMD_ObjectMgr, &ev, somdObjectId);
...
carClassName = _somdTargetGetClassName(car, &ev);
```

SOMDObject

File stem: somdobj

Base

SOMObject

Metaclass

SOMClass

Ancestor Classes

SOMObject

Description

The **SOMDObject** class implements the methods that can be applied to all CORBA object references: e.g., **get_implementation**, **get_interface**, **is_nil**, **duplicate**, and **release**. In the CORBA 1.1 specification, these methods are described in Chapter 8.

In DSOM, there is also another derivation of this class: **SOMDClientProxy**. This subclass inherits the implementation of **SOMDObject**, but extends it by overriding **somDispatch** with a "remote dispatch" method, and caches the binding to the server process. Whenever a remote object is accessed, it is represented in the client process by a **SOMDClientProxy** object.

New methods

The following list shows all the SOMDObject methods.

- create_request
- create_request_args*
- duplicate
- get_implementation
- get_interface
- is_constant*
- is_nil
- is_proxy*
- is_SOM_ref*
- release

(***** These methods were added by DSOM to supplement the published CORBA 1.1 interfaces.)

Overridden methods

The following list shows all the methods overridden by the SOMDObject class. These methods are overridden in order to modify the behavior defined by an ancestor class.

- somInit
- somUninit
- somDumpSelfInt

create_request

This method creates a request to execute a particular operation on the referenced object.

Syntax

```
ORBStatus create_request  (SOMDObject receiver, Environment *env,
                           Context ctx, Identifier operation,
                           NVList arg_list, NamedValue result,
                           Request request, Flags req_flags)
```

Parameters

receiver (SOMDObject)
A pointer to a **SOMDObject** object.

env (Environment *)
A pointer to the **Environment** structure for the method caller.

ctx (Context)
A pointer to the **Context** object of the requested operation.

operation (Identifier)
The name of the operation to be performed on the target object, *receiver*.

arg_list (NVList)
A pointer to a list of arguments (**NVList**). If this argument is NULL, the argument list can be assembled by repeated calls to the **add_arg** method on the **Request** object created by calling this method.

result (NamedValue)
A pointer to a **NamedValue** structure where the result of applying *operation* to *receiver* should be stored.

request (Request)
A pointer to storage for the address of the created **Request** object.

req_flags (Flags)
A **Flags** bitmask (unsigned long) that may contain the following flag value:

OUT_LIST_MEMORY	Indicates that any out-arg memory is associated with the argument list. When the list structure is freed, any associated out-arg memory is also freed. If OUT_LIST_MEMORY is specified, an argument list must also have been specified on the **create_request** call.

Returns

rc (ORBStatus)
Returns an **ORBStatus** value as the status code for the request.

Remarks

The **create_request** method creates a request to execute a particular operation on the referenced object. For more information on the **create_request** call, see CORBA 1.1 page 109.

In DSOM, this method is meaningful only when invoked on a **SOMDClientProxy** object. If invoked on a **SOMDObject** which is not a client proxy, an exception is returned.

Original Class

SOMDObject

Related Methods

Methods

- **create_request_args**
- **create_list**
- **create_operation_list**

Example Code

```
#include  <somd.h>
#include  <repostry.h>
#include  <intfacdf.h>
#include  <foo.h>   /* provided by user */

/* assume following method declaration in interface Foo:
 *      long methodLong  (in long inLong,inout long inoutLong);
 * then the following code builds a request to execute the call:
 *      result = methodLong(fooObj, &ev, 100,200);
 *using the DII.
 */
Environment ev;
OperationDef opdef;
Description desc;
OperationDescription *opdesc;
NVList arglist;
long rc;
long value1 = 100;
long value2 = 200;
Foo fooObj;
Request reqObj;
NamedValue result;
Identifier name;
TypeCode tc;
void *dummy;
long dummylen;
Flags flags;

/* Get the OperationDef from the Interface Repository. */
opdef = _lookup_id(SOM_InterfaceRepository,
                   &ev, "Foo::methodLong");

/* Create a NamedValue list for the operation. */
rc= _create_operation_list
                (SOMD_ORBObject, &ev, opdef, &arglist);

/* Insert arg1 info into arglist */
_get_item(arglist, &ev,
          0, &name, &tc, &dummy, &dummylen, &flags);
_set_item(arglist,&ev,0, name, tc, &value1, sizeof(long), flags);

/* Insert arg2 info into arglist */
_get_item(arglist, &ev,
          1, &name, &tc, &dummy, &dummylen, &flags);
_set_item(arglist,&ev,1, name, tc, &value2, sizeof(long), flags);
```

create_request

```
/* Get the operation description structure. */
desc = _describe(opdef, &ev);
opdesc = (OperationDescription *) desc.value._value;

/* Fill in the TypeCode field for result. */
result.argument._type = opdesc->result;

/* Finally, create the Request, reqObj */
rc = _create_request(fooObj, &ev, (Context *)NULL, "methodLong",
                                  arglist, &result, &reqObj, (Flags)0);
```

create_request_args

This method creates an argument list appropriate for the specified operation.

Syntax

```
ORBStatus create_request_args (SOMDObject receiver,
        Environment *env, Identifier operation, NVList arg_list,
        NamedValue result)
```

Parameters

receiver (SOMDObject)
A pointer to the **SOMDObject** object to create the request.

env (Environment *)
A pointer to the **Environment** structure for the method caller.

operation (Identifier)
The Identifier of the operation for which the argument list is being created.

arg_list (NVList)
A pointer to the location where the method will store a pointer to the resulting argument list.

result (NamedValue)
A pointer to the **NamedValue** structure which will be used to hold the result. The *result*'s type field is filled in with the TypeCode of the expected result.

Returns

rc (ORBStatus)
Returns an **ORBStatus** value representing the return code of the request.

Remarks

The **create_request_args** method creates the appropriate *arg_list* (**NVList**) for the specified operation. It is similar in function to the **create_operation_list** method. Its value is that it also creates the result structure whereas **create_operation_list** does not.

create_request_args

In DSOM, this method is meaningful only when invoked on a **SOMDClientProxy** object. If invoked on a **SOMDObject** which is not a client proxy, an exception is returned.

Original Class

SOMDObject

Related Methods

Methods

- **duplicate**
- **release**
- **create_request**
- **create_operation_list**

Example Code

```
#include  <somd.h>
#include  <repostry.h>
#include  <intfacdf.h>
#include  <foo.h>   /* provided by user */

/* assume following method declaration in interface Foo:
 *      long methodLong  (in long inLong,inout long inoutLong);
 * then the following code builds a request to execute the call:
 *      result = methodLong(fooObj, &ev, 100,200);
 * using the DII.
 */

Environment ev;
OperationDef opdef;
Description desc;
OperationDescription *opdesc;
NVList arglist;
long rc;
long value1 = 100;
long value2 = 200;
Foo fooObj;
Request reqObj;
NamedValue result;
Identifier name;
TypeCode tc;
void *dummy;
long dummylen;
Flags flags;

/* Get the OperationDef from the Interface Repository. */
opdef = _lookup_id(SOM_InterfaceRepository,
                   &ev, "Foo::methodLong");
/* Create a NamedValue list for the operation. */
rc= _create_request_args(fooObj, &ev,
                         "methodLong", &arglist, &result);

/* Insert arg1 info into arglist */
_get_item(arglist, &ev,
          0, &name, &tc, &dummy, &dummylen, &flags);
_set_item(arglist,&ev,0, name, tc, &value1, sizeof(long), flags);
```

create_request_args

```
/* Insert arg2 info into arglist */
_get_item(arglist, &ev,
          1, &name, &tc, &dummy, &dummylen, &flags);
_set_item(arglist,&ev,1, name, tc, &value2, sizeof(long), flags);

/* Finally, create the Request, reqObj */
rc = _create_request(fooObj, &ev, (Context *)NULL, "methodLong",
                                  arglist, &result, &reqObj, (Flags)0);
```

duplicate

This method makes a duplicate of an object reference.

Syntax

```
SOMDObject duplicate  (SOMDObject  receiver, Environment  *env)
```

Parameters

receiver (SOMDObject)
A pointer to a **SOMDObject** object.

env (Environment *)
A pointer to the **Environment** structure for the method caller.

Returns

rc (SOMDObject)
Returns a **SOMDObject** that is a duplicate of the *receiver*. *Ownership* of the returned object is transferred to the caller.

Remarks

The **duplicate** method makes a duplicate of the object reference. The **release** method should be called to free the object.

Original Class

SOMDObject

Related Methods

Methods

- **release**
- **create**
- **create_constant**
- **create_SOM_ref**

duplicate

Example Code

```
#include  <somd.h>

Environment ev;
SOMObject obj;
SOMDObject objref1, objref2;
...
objref1 = _create_SOM_ref(SOMD_SOMOAObject, &ev, obj);
objref2 = _duplicate(objref1,&ev);
...
_release(objref2,&ev);
```

get_implementation

This method returns the implementation definition for the referenced object.

Syntax

```
ImplementationDef get_implementation  (SOMDObject  receiver,
                                       Environment  *env)
```

Parameters

receiver (SOMDObject)
A pointer to a **SOMDObject** object.

env (Environment *)
A pointer to the **Environment** structure for the method caller.

Returns

rc (ImplementationDef)
Returns the **ImplementationDef** object for the *receiver*. *Ownership* of the returned object is transferred to the caller.

Remarks

The **get_implementation** method returns the implementation definition object for the referenced object.

Original Class

SOMDObject

Related Methods

Methods

- **get_interface**

Example Code

```
#include  <somd.h>

long flags;
Environment ev;
SOMDObject objref;
ImplementationDef impldef;
...
impldef = _get_implementation(objref,&ev);
flags = __get_impl_flags(impldef,&ev);
```

get_interface

This method returns the interface definition object for the referenced object.

Syntax

```
InterfaceDef get_interface  (SOMDObject  receiver, Environment  *env)
```

Parameters

receiver (SOMDObject)
A pointer to a **SOMDObject** object.

env (Environment *)
A pointer to the **Environment** structure for the method caller.

Returns

rc (InterfaceDef)
Returns a pointer to the **InterfaceDef** object associated with the reference *receiver*. *Ownership* of the **InterfaceDef** object is passed to the caller.

Remarks

The **get_interface** method returns the interface definition object for the referenced object.

Original Class

SOMDObject

Related Methods

Methods

- **get_implementation**

Example Code

```
#include  <somd.h>
#include  <repostry.h>
#include  <intfacdf.h>

Environment ev;
SOMDObject objref;
InterfaceDef intf;
...
intf = _get_interface(objref,&ev);
```

is_constant

This method tests to see if the object reference is a constant (i.e., its **ReferenceData** is a constant value associated with the reference).

Syntax

```
boolean is_constant   (SOMDObject  receiver, Environment  *env)
```

Parameters

receiver (SOMDObject)
A pointer to a **SOMDObject** object.

env (Environment *)
A pointer to the **Environment** structure for the method caller.

Returns

rc (boolean)
Returns TRUE if the object reference was generated by **create_constant**, Otherwise, **is_constant** returns FALSE.

Remarks

The **is_constant** method tests to see if the object reference was created using the **create_constant** method in the **SOMOA** class.

Related Methods

Methods

- **create**
- **create_constant**
- **is_nil**
- **is_proxy**
- **is_SOM_ref**

Example Code

```
#include  <somd.h>

Environment ev;
SOMDObject objref;
...

/* This code might be part of the code
 * that overrides the somdSOMObjFromRef method, i.e.
 * in an implementation of a subclass of SOMDServer called
 * myServer
 */

if (_is_constant(objref, &ev))
id = _get_id(objref, &ev);

...
```

is_nil

This method tests to see if the object reference is nil.

Syntax

```
boolean is_nil  (SOMDObject  receiver, Environment  *env)
```

Parameters

receiver (SOMDObject)
A pointer to a **SOMDObject** object.

env (Environment *)
A pointer to the **Environment** structure for the method caller.

Returns

rc (boolean)
Returns TRUE if the object reference is empty. Otherwise, **is_nil** returns FALSE.

Remarks

The **is_nil** method tests to see if the specified object reference is nil.

Related Methods

Methods

- **create**
- **is_constant**
- **is_proxy**
- **is_SOM_ref**

Example Code

```
#include  <somd.h>
Environment ev;
SOMDObject objref;
SOMObject somobj;
...
/* This code might be part of the code
 * that overrides the somdSOMObjFromRef method, i.e.
 * in an implementation of a subclass of SOMDServer called
 * myServer
 */
if (_is_nil(objref, &ev) &bxv.&bxv.
    _somIsA(objref, SOMDClientProxyNewClass(0, 0)) &bxv.&bxv.
    _is_SOM_ref(objref, &ev)) {
  somobj = myServer_parent_SOMDServer_somdSOMObjFromRef
           (somSelf, &ev, objref);
}
else {
  /* do the myServer-specific stuff to create/find somobj here */
}
return somobj;
```

is_proxy

This method tests to see if the object reference is a proxy.

Syntax

```
boolean is_proxy   (SOMDObject  receiver, Environment  *env)
```

Parameters

receiver (SOMDObject)
A pointer to a **SOMDObject** object.

env (Environment *)
A pointer to the **Environment** structure for the method caller.

Returns

rc (boolean)
Returns TRUE if the object reference is a proxy object. Otherwise, **is_proxy** returns FALSE.

Remarks

The **is_proxy** method tests to see if the specified object reference is a proxy object.

Original Class

SOMDObject

Related Methods

Methods

- **is_nil**
- **is_constant**
- **is_SOM_ref**
- **string_to_object**

Example Code

```
#include  <somd.h>

SOMDObject objref;
Environment ev;
Context ctx;
NVlist arglist;
NamedValue result;
Request reqObj;
...
if (_is_proxy(objref, &ev)) {
    /* create a remote request for target object */
...
   rc = _create_request(obj, &ev, ctx,
                        "testMethod", arglist, &result, &reqObj,
                        (Flags)0);
}
...
```

is_SOM_ref

This method tests to see if the object reference is a simple reference to a SOM object.

Syntax

```
boolean is_SOM_ref  (SOMDObject  receiver, Environment  *env)
```

Parameters

receiver (SOMDObject)
A pointer to a **SOMDObject** object.

env (Environment *)
A pointer to the **Environment** structure for the method caller.

Returns

rc (boolean)
Returns TRUE if the object reference is a simple (transient) reference to a SOM object. Otherwise, **is_SOM_ref** returns FALSE.

Remarks

The **is_SOM_ref** method tests to see if the specified object reference is a simple (transient) reference to a SOM object.

Original Class

SOMDObject

Related Methods

Methods

- **create_SOM_ref**
- **get_SOM_object**
- **is_proxy**
- **is_nil**
- **is_constant**

Example Code

```
#include  <somd.h>

SOMDObject objref;
Environment ev;
SOMObject obj;
...
if (_is_SOM_ref(objref, &ev))
    /* we know objref is a simple reference, so we can ... */
    obj = _get_SOM_object(SOMD_SOMOAObject, &ev, objref);
...
```

release

This method releases the memory associated with the specified object reference.

Syntax

```
void release  (SOMDObject  receiver, Environment  *env)
```

Parameters

receiver (SOMDObject)
A pointer to a **SOMDObject** object.

env (Environment *)
A pointer to the **Environment** structure for the method caller.

Returns

rc (void)

Remarks

The **release** method releases the memory associated with the object reference.

Original Class

SOMDObject

Related Methods

Methods

- **duplicate**
- **somdReleaseObject**
- **somdReleaseResources**
- **somdProxyFree**
- **create**
- **create_constant**
- **create_SOM_ref**

release

Example Code

```
#include  <somd.h>

SOMDObject objref;
Environment ev;
SOMObject obj;
...
objref = _create_SOM_ref(SOMD_SOMOAObject, &ev, obj);
...
_release(objref, &ev);
```

SOMDObjectMgr

File stem: somdom

Base

ObjectMgr

Metaclass

SOMMSingleInstance

Ancestor Classes

ObjectMgr
SOMObject

Description

The **SOMDObjectMgr** class is derived from **ObjectMgr** class and provides the DSOM implementations for the **ObjectMgr** methods.

Attributes

Listed below is an available **SOMDObjectMgr** attribute, with its corresponding type in parentheses, followed by a description of its purpose:

somd21somFree (boolean)
Determines whether or not **somFree**, when invoked on a proxy object, will free the proxy object along with the remote object. The default value is FALSE, indicating that only the remote object will be freed when **somFree** is invoked on a proxy object. Setting this attribute to TRUE as part of client-program initialization, for example,

```
__set_somd21somdFree(SOMD_ObjectMgr, ev, TRUE);
```

has the effect that all subsequent invocations of **somFree** on proxy objects will free both the remote object and the proxy.

New methods

The following list shows all the SOMDObjectMgr methods.

- somdFindAnyServerByClass*
- somdFindServer*
- somdFindServersByClass*
- somdFindServerByName*

(* This class and its methods were added by DSOM to supplement the published CORBA 1.1 interfaces.)

Overridden methods

The following list shows all the methods overridden by the SOMDObjectMgr class. These methods are overridden in order to modify the behavior defined by an ancestor class.

- somdDestroyObject
- somdGetIdFromObject
- somdGetObjectFromId
- somdNewObject
- somdReleaseObject
- somInit

somdFindAnyServerByClass

This method finds a server capable of creating the specified object.

Syntax

```
SOMDServer somdFindAnyServerByClass  (SOMDObjectMgr  receiver,
                                      Environment  *env,
                                      Identifier  objclass)
```

Parameters

receiver (SOMDObjectMgr)
A pointer to a **SOMDObjectMgr** object.

env (Environment *)
A pointer to the **Environment** structure for the method caller.

objclass (Identifier)
An **Identifier** specifying the class of the object the server needs to be able to create.

Returns

rc (SOMDServer)
Returns a pointer to a **SOMDServer** proxy. If no server can be found in the Implementation Repository that implements the specified class, NULL is returned.

Remarks

The **somdFindAnyServerByClass** method finds a server capable of creating an object of the specified type with the specified properties.

Original Class

SOMDObjectMgr

Related Methods

Methods

- **SomdFindServersByClass**
- **somdFindServer**
- **somdFindServerByName**

Example Code

```
#include  <somd.h>
#include  <stack.h>   /* provided by user */

Stack stk;
Environment ev;
SOMDServer server;

SOM_InitEnvironment(&ev);
SOMD_Init(&ev);
StackNewClass(0,0);
server =
  _somdFindAnyServerByClass(SOMD_ObjectMgr, &ev, "Stack");
stk = _somdCreateObj(server, &ev, "Stack", "");
...
_somdDestroyObject(SOMD_ObjectMgr, &ev, stk);
```

somdFindServer

This method finds a server, given its ImplementationDef ID.

Syntax

```
SOMDServer somdFindServer  (SOMDObjectMgr  receiver,
                            Environment  *env, ImplId  serverid)
```

Parameters

receiver (SOMDObjectMgr)
A pointer to a **SOMDObjectMgr** object.

env (Environment *)
A pointer to the **Environment** structure for the method caller.

serverid (ImplId)
An **ImplId** string which identifies the ImplementationDef of the desired server.

Returns

rc (SOMDServer)
Returns a pointer to a **SOMDServer** proxy.

Remarks

The **somdFindServerByName** method finds a server capable of creating an object of the specified type with the specified properties.

Original Class

SOMDObjectMgr

Related Methods

Methods

- **somdFindServerByName**
- **somdFindServersByClass**
- **somdAnyFindServerByClass**

Example Code

```
#include  <somd.h>
#include  <stack.h>   /* provided by user */

Stack stk;
Environment ev;
SOMDServer server;
ImplId implid;

SOM_InitEnvironment(&ev);
SOMD_Init(&ev);
StackNewClass(0,0);
server = _somdFindServer(SOMD_ObjectMgr, &ev, implid);
stk = _somdCreateObj(server, &ev, "Stack", "");
...
_somdDestroyObject(SOMD_ObjectMgr, &ev, stk);
```

somdFindServerByName

This method finds a server given its **ImplementationDef** name (alias).

Syntax

```
SOMDServer somdFindServerByName  (SOMDObjectMgr  receiver,
                                  Environment  *env,
                                  string  servername)
```

Parameters

receiver (SOMDObjectMgr)
A pointer to a **SOMDObjectMgr** object.

env (Environment *)
A pointer to the **Environment** structure for the method caller.

servername (string)
A **string** which specifies the name of the **ImplementationDef** of the desired server.

Returns

rc (SOMDServer)
Returns a pointer to a **SOMDServer** proxy.

Remarks

The **somdFindServerByName** method finds a server with the specified name.

Original Class

SOMDObjectMgr

Related Methods

Methods

- **somdFindServer**
- **somdFindServersByClass**
- **somdAnyFindServerByClass**

Example Code

```
#include  <somd.h>
#include  <stack.h>   /* provided by user */

Stack stk;
Environment ev;
SOMDServer server;

SOM_InitEnvironment(&ev);
SOMD_Init(&ev);
StackNewClass(0,0);
server =
       _somdFindServerByName(SOMD_ObjectMgr, &ev, "stackServer");
stk = _somdCreateObj(server, &ev, "Stack", "");
...
_somdDestroyObject(SOMD_ObjectMgr, &ev, stk);
```

somdFindServersByClass

This method finds all servers capable of creating a particular object.

Syntax

```
sequence<SOMDServer> somdFindServersByClass
                        (SOMDObjectMgr receiver, Environment *env,
                        Identifier objclass)
```

Parameters

receiver (SOMDObjectMgr)
A pointer to a **SOMDObjectMgr** object.

env (Environment *)
A pointer to the **Environment** structure for the method caller.

objclass (Identifier)
An **Identifier** representing the type of the object the server needs to be able to create.

Returns

rc (sequence<SOMDServer>)
Returns a sequence of **SOMDServer** objects capable of creating the specified object.

Remarks

The **somdFindServersByClass** method finds all servers capable of creating a particular object with the specified properties.

Original Class

SOMDObjectMgr

Related Methods

Methods

- **somdFindServer**
- **somdFindServerByName**
- **somdFindAnyServerByClass**

Example Code

```
#include  <somd.h>
#include  <stack.h>   /* provided by user */

Stack stk;
Environment ev;
sequence(SOMDServer) servers;
SOMDServer server;
SOMDServer chooseServer(sequence(SOMDServer) servers);

SOM_InitEnvironment(&ev);
SOMD_Init(&ev);
StackNewClass(0,0);
servers = _somdFindServersByClass(SOMD_ObjectMgr, &ev, "Stack");
server = chooseServer(servers);
stk = _somdCreateObj(server, &ev, "Stack", "");
...
_somdDestroyObject(SOMD_ObjectMgr, &ev, stk);
```

SOMDServer

File stem: somdserv

Base

SOMObject

Metaclass

SOMMSingleInstance

Ancestor Classes

SOMObject

Description

The **SOMDServer** class is a base class that defines and implements methods for managing objects in a DSOM server process. This includes methods for the creation and deletion of SOM objects, and for getting the SOM class object for a specified class. The **SOMDServer** class also defines and implements methods for the mapping between object references (**SOMDObject**s) and SOM objects, and dispatching methods on objects.

Application-specific methods for managing application objects can be introduced in subclasses of **SOMDServer**.

New methods

The following list shows all the SOMDServer methods.

- somdCreateObj*
- somdDeleteObj*
- somdDispatchMethod*
- somdGetClassObj*
- somdObjReferencesCached*
- somdRefFromSOMObj*
- somdSOMObjFromRef*

(* This class and its methods were added by DSOM to supplement the published CORBA 1.1 interfaces.)

somdCreateObj

This method creates an object of the specified class.

Syntax

```
SOMObject somdCreateObj  (SOMDServer  receiver, Environment  *env,
                          Identifier  objclass, string  hints)
```

Parameters

receiver (SOMDServer)
A pointer to a **SOMDServer** object capable of creating a instance of the specified class.

env (Environment *)
A pointer to the **Environment** structure for the method caller.

objclass (Identifier)
The class of the object for which an instance is to be created.

hints (string)
A **string** which may optionally be used to specify special creation options.

Returns

rc (SOMObject)
Returns a **SOMObject** of the class specified by *objclass*.

Remarks

The **somdCreateObj** method creates an object of the specified class.

Original Class

SOMDServer

Example Code

```
#include  <somd.h>
#include  <stack.h>   /* provided by user */

Stack stk;
Environment ev;
SOMDServer server;

SOM_InitEnvironment(&ev);
SOMD_Init(&ev);
StackNewClass(0,0);
server =
  _somdFindServerByName(SOMD_ObjectMgr, &ev,"stackServer");
stk = _somdCreateObj(server, &ev, "Stack", "");
...
_somdDestroyObject(SOMD_ObjectMgr, &ev, stk);
```

somdDeleteObj

This method deletes the specified object.

Syntax

```
void somdDeleteObj  (SOMDServer  receiver, Environment  *env,
                     SOMObject  somobj)
```

Parameters

receiver (SOMDServer)
A pointer to a **SOMDServer** object.

env (Environment *)
A pointer to the **Environment** structure for the method caller.

somobj (SOMObject)
An object "managed" by the server object.

Returns

rc (void)

Remarks

The **somdDeleteObj** method deletes the specified object.

Original Class

SOMDServer

Example Code

```
#include  <somd.h>
#include  <stack.h>   /* provided by user */

Stack stk;
Environment ev;
SOMDServer server;

SOM_InitEnvironment(&ev);
SOMD_Init(&ev);
StackNewClass(0,0);
server =
  _somdFindServerByName(SOMD_ObjectMgr, &ev,"stackServer");
stk = _somdCreateObj(server, &ev, "Stack", "");
...
_somdDeleteObj(server, &ev, stk);
```

somdDispatchMethod

This method dispatches a method on the specified SOM object.

Syntax

```
void somdDispatchMethod  (SOMDServer  receiver, Environment  *env,
                          SOMObject  somobj, somToken  retValue,
                          somId  methodId, va_list  ap)
```

Parameters

receiver (SOMDServer)
A pointer to a **SOMDServer** object.

env (Environment *)
A pointer to the **Environment** structure for the method caller.

somobj (SOMObject)
A pointer to an object "managed" by the server object.

retValue (somToken)
A pointer to the storage area allocated to hold the method result value, if any.

methodId (somId)
A **somId** for the name of the method which is to be dispatched.

ap (va_list)
A pointer to a **va_list** array of arguments to the method call.

Returns

rc (void)
Returns a result, if any, in the storage whose address is in *retValue*.

Remarks

The **somdDispatchMethod** method is used to intercept method calls on objects in a server. When a request arrives, the request parameters are extracted from the message, and the target object is resolved. Then, the **SOMOA** dispatches the method call on the target object using the **somdDispatchMethod** method.

The default implementation will call **somDispatch** on the target object with the parameters as specified. This method can be overridden to intercept and process the method calls before they are dispatched.

Original Class

SOMDServer

Example Code

```
#include  <somd.h>

/* overridden somdDispatchMethod */
void somdDispatchMethod(SOMDServer *somself, Environment *ev,
                                    SOMObject *somobj, somToken *retValue,
                                    somId methodId, va_list ap)
{
   printf("dispatching %s on %x\n", SOM_StringFromId(methodId),
         somobj);
   SOMObject_somDispatch(somobj, ev, retValue, methodId, ap);
}
```

somdGetClassObj

This method creates a class object for the specified class.

Syntax

```
SOMClass somdGetClassObj   (SOMDServer  receiver, Environment  *env,
                            Identifier  objclass)
```

Parameters

receiver (SOMDServer)
A pointer to a **SOMDServer** object.

env (Environment *)
A pointer to the **Environment** structure for the method caller.

objclass (Identifier)
An identifier specifying the type of the class object to be created.

Returns

rc (SOMClass)
Returns a SOMClass object of the type specified.

Remarks

The **somdGetClassObj** method creates a class object of the specified type.

Original Class

SOMDServer

Example Code

```
#include  <somd.h>
#include  <stack.h>   /* provided by user */

SOMClass stkclass;
Environment ev;
SOMDServer server;

SOM_InitEnvironment(&ev);
SOMD_Init(&ev);
StackNewClass(0,0);
server =
  _somdFindServerByName(SOMD_ObjectMgr, &ev,"stackServer");
stkclass =  _somdGetClassObj(server, &ev, "Stack", "");
```

somdObjReferencesCached

This method indicates whether a server object retains ownership of the object references it creates via the **somdRefFromSOMObj** method.

Syntax

```
boolean somdObjReferencesCached  (SOMDServer  receiver,
                                  Environment  *env)
```

Parameters

receiver (SOMDServer)
A pointer to an object of class **SOMDServer**.

env (Environment *)
A pointer to the **Environment** structure for the calling method.

Returns

rc (boolean)

FALSE	Returns FALSE by default.
TRUE	Overriding implementations may return TRUE to indicate that a subclass of **SOMDServer** implements object reference caching.

Remarks

This method indicates whether a server object retains ownership of the object references it creates via the **somdRefFromSOMObj** method. The default implementation returns FALSE, meaning that the server turns over ownership of the object references it creates to the caller. Subclasses of **SOMDServer** that implement object reference caching should override this method to return TRUE.

Original Class

SOMDServer

Related Methods

Methods

- **somdRefFromSOMObj**

Example Code

```
SOMDobject objref;
objref = _somdRefFromSOMObj(serverObj, ev, myobj);
...
/* code to use objref */
...
if (!_somdObjReferencesCached(serverObj, ev))
    _release(objref, ev);
```

somdRefFromSOMObj

This method returns an object reference corresponding to the specified SOM object.

Syntax

```
SOMDObject somdRefFromSOMObj  (SOMDServer  receiver,
                               Environment  *env,
                               SOMObject  somobj)
```

Parameters

receiver (SOMDServer)
A pointer to a **SOMDServer** object.

env (Environment *)
A pointer to the **Environment** structure for the method caller.

somobj (SOMObject)
A pointer to the SOM object for which a DSOM reference is to be created.

Returns

rc (SOMDObject)
Returns a DSOM reference (that is, a **SOMDObject**) for the SOM object specified.

Remarks

The **somdRefFromSOMObj** method creates a simple (transient) reference to a SOM object. This method is called by **SOMOA** as part of converting the results of a local method call into a result message for a remote client.

By default the **somdRefFromSOMObj** method turns over ownership of the object reference it creates to the caller. However, if a subclass of **SOMDServer** overrides **somdRefFromSOMObj** to implement object reference caching, then that subclass should also override the method **somdObjReferencesCached** to report that caching by returning TRUE.

Original Class

SOMDServer

Related Methods

Methods

- **somdObjReferencesCached**

Example Code

```
#include  <somd.h>
#include  <stack.ih> /* user-generated */

SOMDObject objref;
Environment ev;
SOMObject obj;
...
/* myServer specific code up here */
...
/* one might want to make this call as part of the code
 * that overrides the somdRefFromSOMObj method, i.e.
 * in an implementation of a subclass of SOMDServer called
 * myServer
 */
objref =
 myServer_parent_SOMDServer_somdRefFromSOMObj(somSelf, &ev, obj);
```

somdSOMObjFromRef

This method returns the SOM object corresponding to the specified object reference.

Syntax

```
SOMObject somdSOMObjFromRef  (SOMDServer  receiver,
                              Environment  *env,
                              SOMDObject  objref)
```

Parameters

receiver (SOMDServer)
A pointer to a **SOMDServer** object.

env (Environment *)
A pointer to the **Environment** structure for the method caller.

objref (SOMDObject)
A pointer to the DSOM object reference to the SOM object.

Returns

rc (SOMObject)
Returns the SOM object associated with the supplied DSOM reference.

Remarks

The **somdSOMObjFromRef** method returns the SOM object associated with the DSOM object reference, *objref*. This method is called by **SOMOA** as part of converting a remote request into a local method call on an object.

Original Class

SOMDServer

Example Code

```
#include  <somd.h>
#include  <stack.ih>  /* user-generated */

SOMDObject objref;
Environment ev;
SOMObject obj;
...
/* myServer specific code up here */
...
/* one might want to make this call as part of the code
 * that overrides the somdRefFromSOMObj method, i.e.
 * in an implementation of a subclass of SOMDServer called
 * myServer
 */
obj =
myServer_parent_SOMDServer_somdSOMObjFromRef(somSelf,&ev,objref);
```

SOMDServerMgr

File stem: somoa

Base

SOMObject

Metaclass

SOMClass

Ancestor Classes

SOMOBject

Description

The **SOMDServerMgr** class provides a programmatic interface to manager server processes. At present, the server processes that can be managed are limited to those present in the Implementation Repository. The choice of Implementation Repository is determined by the environment variable SOMDDIR.

New methods

The following list shows all the SOMDServerMgr methods.

- somdDisableServer
- somdEnableServer
- somdIsServerEnabled
- somdListServer
- somdRestartServer
- somdShutdownServer
- somdStartServer

somdDisableServer

This method disables a server process from starting until it is explicitly enabled again.

Syntax

```
ORBStatus somdDisableServer  (SOMDServerMgr  receiver,
                              Environment  *env, string  server_alias)
```

Parameters

receiver (SOMDServerMgr)
A pointer to an object of class **SOMDServerMgr**.

env (Environment *)
A pointer to the **Environment** structure for the calling method.

server_alias (string)
The implementation alias of the server to be disabled.

Returns

rc (ORBStatus)
Returns 0 for success or a DSOM error code for failure.

Remarks

The **somdDisableServer** method disables the server process associated with the server alias. Once a server process has been disabled, it cannot be restarted until it is explicitly enabled again. Initially, all server processes are enabled by default. Note: If the server process to be disabled is currently running, then it is first stopped before disabling. If the method is unsuccessful in stopping the server, the disable method fails.

Original Class

SOMDServerMgr

Related Methods

Methods

- **somdEnableServer**

somdDisableServer

Example Code

```
#include <somd.h>
#include <servmgr.h>

SOMDServerMgr servmgr;
string  server_alias = "MyServer";
ORBStatus rc;
Environment e;

SOM_InitEnvironment(&e);
SOMD_Init(&e);
servmgr = SOMDServerMgrNew();
rc = _somdDisableServer(servmgr, &e, server_alias);
```

somdEnableServer

This method enables a server process so that it can be started when required. Initially, all server processes are enabled by default.

Syntax

```
ORBStatus somdEnableServer  (SOMDServerMgr  receiver,
                             Environment  *env, string  server_alias)
```

Parameters

receiver (SOMDServerMgr)
A pointer to an object of class **SOMDServerMgr**.

env (Environment *)
A pointer to the **Environment** structure for the calling method.

server_alias (string)
The implementation alias of the server to be enabled.

Returns

rc (ORBStatus)
Returns 0 for success or a DSOM error code for failure.

Remarks

The **somdEnableServer** method enables a server process associated with the server alias. Initially, all server processes are enabled by default. Server processes can be disabled by using the **somdDisableServer** method.

Original Class

SOMDServerMgr

Related Methods

Methods

- **somdDisableServer**

Example Code

```
SOMDServerMgr servmgr;
string  server_alias = "MyServer";
ORBStatus rc;
Environment e;

SOM_InitEnvironment(&e);
SOMD_Init(&e);
servmgr = SOMDServerMgrNew();

/* disable the server */
rc = _somdDisableServer(servmgr, &e, server_alias);

/*  do some processing */

/* enable the server */
rc = _somdEnableServer(servmgr, &e, server_alias);
```

somdIsServerEnabled

This method determines whether a server process is enabled or not.

Syntax

```
boolean somdIsServerEnabled  (SOMDServerMgr  receiver,
                              Environment  *env,
                              ImlementationDef  impldef)
```

Parameters

receiver (SOMDServerMgr)
A pointer to an object of class **SOMDServerMgr**.

env (Environment *)
A pointer to the **Environment** structure for the calling method.

impldef (ImlementationDef)
A pointer to the **ImplementationDef** object for the server, obtained using the **find_impldef_by_alias** method when it is invoked on the global **SOMD_ImplRepObject**.

Returns

rc (boolean)
Returns TRUE if the server is enabled; otherwise, FALSE is returned.

Remarks

The **somdIsServerEnabled** method returns a **boolean** corresponding to the current state (enabled/disabled) of the server process.

Original Class

SOMDServerMgr

Related Methods

Methods

- **somDisableServer**
- **somEnableServer**

somdIsServerEnabled

Example Code

```
#include <somd.h>
#include <servmgr.h>

SOMDServerMgr servmgr;
ImplementationDef impldef;
string  server_alias = "MyServer";
boolean rc;
Environment e;

SOM_InitEnvironment(&e);
SOMD_Init(&e);

impldef = _find_impldef_by_alias(SOMD_ImplRepObject,
                                 &e, server_alias);
servmgr = SOMDServerMgrNew();

/* if server is disabled then enable it*/
if (!_somdIsServerEnabled(servmgr, &e, impldef))
    rc = _somdEnableServer(servmgr, &e, server_alias)          ;
```

somdListServer

This method queries the state of a server process.

Syntax

```
ORBStatus somdListServer  (SOMDServerMgr  receiver,
                           Environment  *env, string  server_alias)
```

Parameters

receiver (SOMDServerMgr)
A pointer to an object of class **SOMDServerMgr**.

env (Environment *)
A pointer to the **Environment** structure for the calling method.

server_alias (string)
The implementation alias of the server to be listed.

Returns

rc (ORBStatus)
Returns 0 if the server process is running; otherwise, a DSOM error code is returned.

Remarks

The **somdListServer** method is invoked to query the status of the server process associated with the server alias. If the server process is running, the return code will be 0 indicating success.

Original Class

SOMDServerMgr

Example Code

```
#include <somd.h>
#include <servmgr.h>

SOMDServerMgr servmgr;
string  server_alias = "MyServer";
ORBStatus rc;
Environment e;

SOM_InitEnvironment(&e);
SOMD_Init(&e);
servmgr = SOMDServerMgrNew();
rc = _somdListServer(servmgr, &e, server_alias);
if (!rc)                          /* server is running */
    rc = _somdShutdownServer(servmgr, &e, server_alias);
else if (rc == SOMDERROR_ServerNotFound)
                             /* server is not running */
    rc = _somdStartServer(servmgr, &e, server_alias);
```

somdRestartServer

This method restarts a server process.

Syntax

```
ORBStatus somdRestartServer  (SOMDServerMgr  receiver,
                              Environment  *env, string  server_alias)
```

Parameters

receiver (SOMDServerMgr)
A pointer to an object of class **SOMDServerMgr**.

env (Environment *)
A pointer to the **Environment** structure for the calling method.

server_alias (string)
The implementation alias of the server to be restarted.

Returns

rc (ORBStatus)
Returns 0 for success or a DSOM error code for failure.

Remarks

The **somdRestartServer** method is invoked to restart a server process. If the server process currently exists, it will be stopped and started again. If the server process does not exist, a new server process will still be started. If the server process cannot be stopped and/or started for any reason, the method returns a DSOM error code.

Note: If the designated server is registered in the Implementation Repository (on the server's machine) as "nonstoppable" (via the **regimpl** "-n" option), then the method will return an error.

Original Class

SOMDServerMgr

somdRestartServer

Example Code

```
#include <somd.h>
#include <servmgr.h>

SOMDServerMgr servmgr;
string  server_alias = "MyServer";
ORBStatus rc;
Environment e;

SOM_InitEnvironment(&e);
SOMD_Init(&e);
servmgr = SOMDServerMgrNew();
rc = _somdRestartServer(servmgr, &e, server_alias);
```

somdShutdownServer

This method stops a server process.

Syntax

```
ORBStatus somdShutdownServer  (SOMDServerMgr  receiver,
                               Environment  *env, string  server_alias)
```

Parameters

receiver (SOMDServerMgr)
A pointer to an object of class **SOMDServerMgr**.

env (Environment *)
A pointer to the **Environment** structure for the calling method.

server_alias (string)
The implementation alias of the server to be stopped.

Returns

rc (ORBStatus)
Returns 0 for success or a DSOM error code for failure.

Remarks

The **somdShutdownServer** method is invoked to stop a server process. If the server process corresponding to the server alias exists, it will be stopped and a code indicating success is returned.

Note: If the designated server is registered in the Implementation Repository (on the server's machine) as “nonstoppable” (via the **regimpl** “-n” option), then this method will return an error.

Note: On AIX, this method will fail to stop the server process if the process owner executing this method is not the same as that of either the server process or root.

Original Class

SOMDServerMgr

somdShutdownServer

Example Code

```
#include <somd.h>
#include <servmgr.h>

SOMDServerMgr servmgr;
string  server_alias = "MyServer";
ORBStatus rc;
Environment e;

SOM_InitEnvironment(&e);
SOMD_Init(&e);
servmgr = SOMDServerMgrNew();
rc = _somdShutdownServer(servmgr, &e, server_alias);
```

somdStartServer

This method starts a server process.

Syntax

```
ORBStatus somdStartServer  (SOMDServerMgr  receiver,
                            Environment  *env, string  server_alias)
```

Parameters

receiver (SOMDServerMgr)
A pointer to an object of class **SOMDServerMgr**.

env (Environment *)
A pointer to the **Environment** structure for the calling method.

server_alias (string)
The implementation alias of the server to be started.

Returns

rc (ORBStatus)
Returns 0 for success or a DSOM error code for failure.

Remarks

The **somdStartServer** method is invoked to start a server process. If the server process does not exist, the server process is started and the code indicating success is returned. If the server process already exists, then the return code will still indicate success and the server process will be undisturbed.

Original Class

SOMDServerMgr

somdStartServer

Example Code

```
#include <somd.h>
#include <servmgr.h>

SOMDServerMgr servmgr;
string  server_alias = "MyServer";
ORBStatus rc;
Environment e;

SOM_InitEnvironment(&e);
SOMD_Init(&e);
servmgr = SOMDServerMgrNew();
rc = _somdStartServer(servmgr, &e, server_alias);
```

SOMOA

File stem: somoa

Base

BOA

Metaclass

SOMMSingleInstance

Ancestor Classes

BOA

SOMObject

Description

The **SOMOA** class is DSOM's basic object adapter. **SOMOA** is a subclass of the abstract **BOA** class, and provides implementations of all the **BOA** methods. The **SOMOA** class also introduces methods for receiving and dispatching requests on SOM objects. **SOMOA** provides some additional methods for creating and managing object references.

New methods

The following list shows all the SOMOA methods.

- activate_impl_failed*
- change_id*
- create_constant*
- create_SOM_ref*
- execute_next_request*
- execute_request_loop*
- get_SOM_object*

(***** This class and its methods were added by DSOM to supplement the published CORBA 1.1 interfaces.)

Overridden methods

The following list shows all the methods overridden by the SOMOA class. These methods are overridden in order to modify the behavior defined by an ancestor class.

- change_implementation
- create
- deactivate_impl
- deactivate_obj

- dispose
- get_id
- get_principal
- impl_is_ready
- obj_is_ready
- set_exception
- somInit
- somUninit

activate_impl_failed

This message sends a message to the DSOM daemon indicating that a server did not activate.

Syntax

```
void activate_impl_failed  (SOMOA  receiver, Environment  *env,
                            ImplementationDef  implDef, long  rc)
```

Parameters

receiver (SOMOA)
A pointer to the **SOMOA** object that attempted to activate the implementation.

env (Environment *)
A pointer to the **Environment** structure for the method caller.

implDef (ImplementationDef)
A pointer to the **ImplementationDef** object representing the implementation that failed to activate.

rc (long)
A return code designating the reason for failure.

Returns

rc (void)

Remarks

The **activate_impl_failed** method sends a message to the DSOM daemon (**somdd**) indicating that the server did not activate.

Original Class

SOMOA

activate_impl_failed

Example Code

```
#include  <somd.h>   /* needed by all servers */
main(int argc, char **argv)
{
   Environment ev;
   SOM_InitEnvironment(&ev);

   /* Initialize the DSOM run-time environment */
   SOMD_Init(&ev);

   /* Retrieve its ImplementationDef from the Implementation
      Repository by passing its implementation ID as a key */
   SOMD_ImplDefObject =
      _find_impldef(SOMD_ImplRepObject, &ev, argv[1]);

   /* create the SOMOA */
   SOMD_SOMOAObject = SOMOANew();
...
/* suppose something went wrong with server initialization */
...
/* tell the daemon (via SOMOA) that activation failed */
_activate_impl_failed(SOMD_SOMOAObject,
                      &ev, SOMD_ImplDefObject, rc);
```

change_id

This method changes the reference data associated with an object.

Syntax

```
void change_id  (SOMOA  receiver, Environment  *env,
                 SOMDObject  objref, ReferenceData  id)
```

Parameters

receiver (SOMOA)
A pointer to the **SOMOA** object managing the implementation.

env (Environment *)
A pointer to the **Environment** structure for the method caller.

objref (SOMDObject)
A pointer to the **SOMDObject** which identifies the object.

id (ReferenceData)
A pointer to the **ReferenceData** structure representing the object to be created.

Returns

rc (void)

Remarks

The **change_id** changes the ReferenceData associated with the object identified by *objref*. The ReferenceData previously stored in the SOMOA's reference data table is replaced with the value of *id*. The new ID cannot be larger than the maximum size of the original ReferenceData (usually specified as 1024 bytes).

change_id

Example Code

```
#include  <somd.h>
#include  <repostry.h>
#include  <intfacdf.h>

SOMDObject objref;
ReferenceData id1, id2;
InterfaceDef intfdef;
...
objref = _create(SOMD_SOMOAObject, &ev, id1,
         intfdef, SOMD_ImplDefObject);
...
/* change the ReferenceData associated with a SOMDObject */
_change_id(SOMD_SOMOAObject, &ev, objref, id2);
```

create_constant

This method creates a "constant" object reference.

Syntax

```
SOMDObject create_constant  (SOMOA  receiver, Environment  *env,
                             ReferenceData  id, InterfaceDef  intf,
                             ImplementationDef  impl)
```

Parameters

receiver (SOMOA)
A pointer to the **SOMOA** object managing the implementation.

env (Environment *)
A pointer to the **Environment** structure for the method caller.

id (ReferenceData)
A pointer to the **ReferenceData** structure containing application-specific information describing the target object.

intf (InterfaceDef)
A pointer to the **InterfaceDef** object which describes the interface of the target object.

impl (ImplementationDef)
A pointer to the **ImplementationDef** object which describes the application (server) process which implements the target object.

Returns

rc (SOMDObject)
Returns a pointer to a **SOMDObject**. *Ownership* of the new object reference is transferred to the caller.

Remarks

The **create_constant** method is a variant of the **create** method. Like **create**, it creates an object reference for an object with the specified interface and associates the supplied ReferenceData with the object reference. The ReferenceData can later be retrieved using the **get_id** method. Unlike **create**, this method creates a "contant" reference whose ID value cannot be changed. (See the **change_id** method of

SOMOA.) This is because the ID is maintained as a constant part of the object reference state, versus stored in the reference data table for the server.

This method would be used whenever the application prefers not to maintain an object's **ReferenceData** in the server's reference data table.

Original Class

SOMOA

Related Methods

Methods

- **create**
- **create_SOM_ref**
- **dispose**
- **get_id**
- **is_constant**

Example Code

```
#include  <somd.h>
#include  <repostry.h>
#include  <intfacdf.h>

Environment ev;
ReferenceData id;
InterfaceDef intfdef;
SOMDObject objref;
string fname; /* file name to be saved with reference */
...
/* create the id for the reference */
id._maximum = id._length = strlen(fname)+1;
id._buffer = (string) SOMMalloc(strlen(fname)+1);
strcpy(id._buffer,fname);

/* get the interface def object for interface Foo*/
intfdef = _lookup_id(SOM_InterfaceRepository, &ev, "Foo");

objref = _create_constant(SOMD_SOMOAObject,
                 &ev, id, intfdef, SOMD_ImplDefObject);
```

create_SOM_ref

This method creates a simple, transient DSOM reference to a SOM object.

Syntax

```
SOMDObject create_SOM_ref  (SOMOA  receiver, Environment  *env,
                            SOMObject  somobj,
                            ImplementationDef  impl)
```

Parameters

receiver (SOMOA)
A pointer to the **SOMOA** object managing the implementation.

env (Environment *)
A pointer to the **Environment** structure for the method caller.

somobj (SOMObject)
A pointer to the local **SOMObject** to be referenced.

impl (ImplementationDef)
A pointer to the **ImplementationDef** of the calling server process.

Returns

rc (SOMDObject)
Returns a pointer to a **SOMDObject**. *Ownership* of the new object reference is transferred to the caller.

Remarks

The **create_SOM_ref** method creates a simple DSOM reference (**SOMDObject**) for a local SOM object. The reference is "special" in that there is no explicit **ReferenceData** associated with the object. Also, this object reference is only valid while the target SOM object exists.

The **SOMObject** associated with the SOM_ref can be retrieved via the **get_SOM_object** method. The **is_SOM_ref** method of **SOMDObject** can be used to tell if the reference was created using **create_SOM_ref** or not.

create_SOM_ref

Original Class

SOMOA

Related Methods

Methods

- **get_SOM_object**
- **is_SOM_ref**

Example Code

```
#include  <somd.h>

SOMDObject objref;
Environment ev;
SOMObject obj;
...
/* one might want to make this call as part of the code
 * that overrides the somdRefFromSOMObj method, i.e.
 * in an implementation of a subclass of SOMDServer.
 */
objref = _create_SOM_ref(SOMD_SOMOAObject, &ev,
                                                    obj, SOMD_ImplDefObject);
```

execute_next_request

This method receives a request message, executes the request, and returns to the caller.

Syntax

```
ORBStatus execute_next_request (SOMOA  receiver, Environment  *env,
                Flags  waitFlag)
```

Parameters

receiver (SOMOA)
A pointer to the **SOMOA** object managing the implementation.

env (Environment *)
A pointer to the **Environment** structure for the method caller.

waitFlag (Flags)
A **Flags** value (unsigned long) indicating whether the method should block if there is no message pending (SOMD_WAIT) or return with an error (SOMD_NO_WAIT).

Returns

rc (ORBStatus)
Returns an **ORBStatus** value representing the return value for the operation. SOMDERROR_NoMessages is returned if the method is invoked with SOMD_NO_WAIT and no message is available.

Remarks

The **execute_next_request** method receives the next request message, executes the request, and sends the result to the caller.

If the server's **ImplementationDef** indicates the server is multithreaded (the **impl_flags** has the IMPLDEF_MULTI_THREAD flag set), each request will be run by **SOMOA** in a separate thread.

Original Class

SOMOA

execute_next_request

Related Methods

Methods

- **execute_request_loop**

Example Code

```
#include  <somd.h>

/* server initialization code ... */
SOM_InitEnvironment(&ev);

/* signal DSOM that server is ready */
_impl_is_ready(SOMD_SOMOAObject, &ev, SOMD_ImplDefObject);

while (ev._major == NO_EXCEPTION) {
    (void)_execute_next_request(SOMD_SOMOAObject,&ev,SOMD_WAIT );
    /* perform appl-specific code between messages here, e.g.,*/
    numMessagesProcessed++;
}
```

execute_request_loop

This method receives a request message, executes the request, and returns the result to the calling client.

Syntax

```
ORBStatus execute_request_loop (SOMOA  receiver, Environment  *env,
                Flags  waitFlag)
```

Parameters

receiver (SOMOA)
A pointer to the **SOMOA** object managing the implementation.

env (Environment *)
A pointer to the **Environment** structure for the method caller.

waitFlag (Flags)
A **Flags** bitmask (unsigned long) indicating whether the method should block (SOMD_WAIT) or return to the caller (SOMD_NO_WAIT) when there is no request message pending.

Returns

rc (ORBStatus)
May return an **OBJ_ADAPTER** exception which contains an DSOM error code for the operation. SOMDERROR_NoMessages is returned as an **ORBStatus** code if the method is invoked with SOMD_NO_WAIT and no message is pending.

Remarks

The **execute_request_loop** method initiates a loop that waits for a request message, executes the request, and returns the result to the client who invoked the request. When called with the SOMD_WAIT flag, this method loops infinitely (or until an error). When called with the SOMD_NO_WAIT flag, this method loops as long as it finds a request message to process.

The SOMD_NO_WAIT flag is useful when writing event-driven applications where there are event sources other than DSOM requests (e.g., user input, etc.). In this case, DSOM cannot be given exclusive control. Instead, a DSOM event handler can be written using the SOMD_NO_WAIT option, to process all pending requests before returning control to the event manager.

execute_request_loop

If the server's **ImplementationDef** indicates the server is multithreaded (the **impl_flags** has the IMPLDEF_MULTI_THREAD flag set), each request will be run by **SOMOA** in a separate thread (OS/2 only).

See Chapter 9 of the *SOM Programming Guide* for a description of the Event Manager (EMan) framework, for writing event-driven applications.

Original Class

SOMOA

Related Methods

Functions

- **SOMD_RegisterCallback**

Methods

- **execute_next_request**

Example Code

```
#include  <somd.h>

/* server initialization code ... */
...
_impl_is_ready(SOMD_SOMOAObject, &ev, SOMD_ImplDefObject);

/* turn control over to SOMOA */
(void) _execute_request_loop(SOMD_SOMOAObject, &ev,SOMD_WAIT);
```

get_SOM_object

Get the SOM object associated with a simple DSOM reference.

Syntax

```
SOMObject get_SOM_object  (SOMOA  receiver, Environment  *env,
                           SOMDObject  somref)
```

Parameters

receiver (SOMOA)
A pointer to the **SOMOA** object managing the implementation.

env (Environment *)
A pointer to the **Environment** structure for the method caller.

somref (SOMDObject)
A pointer to a **SOMDObject** created by the **create_SOM_ref** method.

Returns

rc (SOMObject)
Returns the SOM object associated with the reference.

Remarks

The **get_SOM_object** method returns the SOM object associated with a reference created by the **create_SOM_ref** method.

Original Class

SOMOA

Related Methods

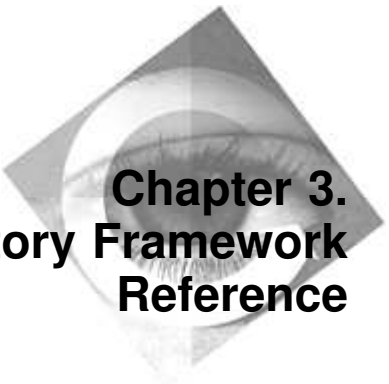
Methods

- create_SOM_ref
- is_SOM_ref

Example Code

```
#include  <somd.h>

SOMDObject objref;
Environment ev;
SOMObject obj;
...
if (_is_SOM_ref(objref, &ev))
    /* we know objref is a simple reference, so we can ... */
    obj = _get_SOM_object(SOMD_SOMOAObject, &ev,objref);
...
```



Chapter 3. Interface Repository Framework Reference

© Copyright IBM Corp. 1994, 1996

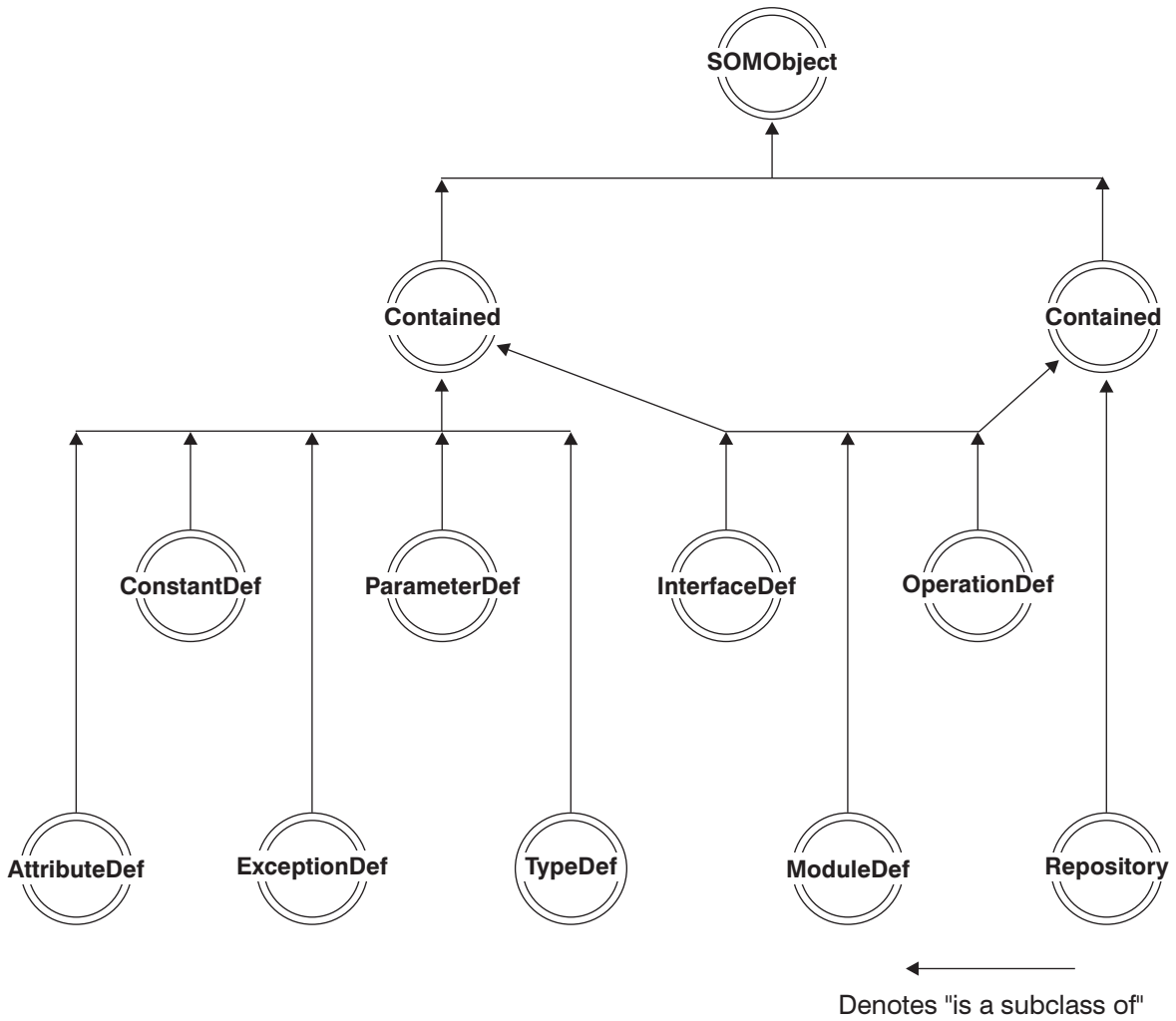


Figure 3. Interface Repository Framework Class Organization

AttributeDef

File stem: attribdf

Base

Contained

Metaclass

SOMClass

Ancestor Classes

Contained

SOMObject

Description

Types

```
enum  AttributeMode   {NORMAL, READONLY};

struct  AttributeDescription   {
    Identifier      name;
    RepositoryId    id;
    RepositoryId    defined_in;
    TypeCode        type;
    AttributeMode   mode;
};
```

The **describe** method, inherited from **Contained,** returns an **AttributeDescription** structure in the "value" member of the **Description** structure (defined in the **Contained** class).

Attributes

Listed below is each available attribute, with its corresponding type in parentheses, followed by a description of its purpose:

type (TypeCode)

The **TypeCode** that represents the type of the **attribute**. The **TypeCode** returned by the "_get_" form of the **type** attribute is contained in the receiving **AttributeDef** object, which retains ownership. Hence, the returned **TypeCode** should not be freed. To obtain a separate copy, use the **TypeCode_copy** operation. The "_set_" form of the attribute makes a private copy of the **TypeCode** you supply, to keep in the receiving object. You retain ownership of the passed **TypeCode**.

mode (AttributeMode)
The AttributeMode of the attribute (NORMAL or READONLY).

New methods

There are currently no new methods defined for the AttributeDef class.

Overridden methods

The following list shows all the methods overridden by the AttributeDef class. These methods are overridden in order to modify the behavior defined by an ancestor class.

- somInit
- somUninit
- somDumpSelf
- somDumpSelfInit
- describe

ConstantDef

File stem: constdef

Base

Contained

Metaclass

SOMClass

Ancestor Classes

Contained

SOMObject

Description

Types

```
struct ConstantDescription   {
Identifier    name;
RepositoryId  id;
RepositoryId  defined_in;
TypeCode      type;
any value;
};
```

The **describe** method, inherited from **Contained,** returns a **ConstantDescription** structure in the "value" member of the **Description** structure (defined in the **Contained** class).

Attributes

Listed below is each available attribute, with its corresponding type in parentheses, followed by a description of its purpose:

type (TypeCode)

The **TypeCode** that represents the type of constant. The **TypeCode** returned by the "_get_" form of the **type** attribute is contained in the receiving **ConstantDef** object, which retains ownership. Hence, the returned **TypeCode** should not be freed. To obtain a separate copy, use the **TypeCode_copy** operation. The "_set_" form of the attribute makes a private copy of the **TypeCode** you supply, to keep in the receiving object. You retain ownership of the passed **TypeCode**.

value (any)

The value of the **constant**.

New methods

There are currently no new methods defined for the ConstantDef class.

Overridden methods

The following list shows all the methods overridden by the ConstantDef class. These methods are overridden in order to modify the behavior defined by an ancestor class.

- somInit
- somUninit
- somDumpSelf
- somDumpSelfInt
- describe

Contained

File stem: containd

Base

SOMObject

Metaclass

SOMClass

Ancestor Classes

SOMObject

Description

Types

```
typedef  string       RepositoryId;
 struct       Description   {
 Identifier   name;
 any          value;
       };
```

Attributes

All attributes of the **Contained** class provide access to information kept within the receiving object. The "_get_" form of the attribute returns a memory reference that is only valid as long as the receiving object has not been freed (using **_somFree**). The "_set_" form of the attribute makes a (deep) copy of your data and places it in the receiving object. You retain ownership of all memory references passed using the "_set_" attributes.

Listed below is each available attribute, with its corresponding type in parentheses, followed by a description of its purpose:

name (Identifier)
A simple name that identifies the **Contained** object within its containment hierarchy. The name may not be unique outside of the containment hierarchy; thus it may require qualification by **ModuleDef** name and/or **InterfaceDef** name.

id (RepositoryId)
The value of the "id" field of the **Contained** object. This is a string that uniquely identifies any object in the IR; thus it needs no qualification. Note that **RepositoryId**s have no relationship to the SOM type **somId**.

defined_in (RepositoryId)

The value of the "defined_in" field of the **Contained** object. This ID uniquely identifies the container where the **Contained** object is defined. Objects without global scope that do not appear within any other object are, by default, placed in the **Repository** object.

somModifiers (sequence <somModifier>)

The somModifiers attribute is a sequence containing all modifiers associated with the object in the "implementation" section of the SOM IDL file where the receiving object is defined.

Note: This attribute is a SOM-unique extension of the Interface Repository; it is not stipulated by the CORBA specification.

New methods

The following list shows all the Contained methods.

- within
- describe

Overridden methods

The following list shows all the methods overridden by the Contained class. These methods are overridden in order to modify the behavior defined by an ancestor class.

- somFree
- somInit
- somUninit
- somDumpSelf
- somDumpSelfInt

describe

This method returns a structure containing information defined in the IDL specification that corresponds to a specified **Contained** object in the Interface Repository.

Syntax

```
Description describe  (Contained  receiver, Environment  *env)
```

Parameters

receiver (Contained)
A pointer to the **Contained** object in the Interface Repository for which a **Description** is needed.

env (Environment *)
A pointer to the **Environment** structure for the caller.

Returns

rc (Description)
Returns a structure of type **Description** containing information defined in the IDL specification of the receiving object.

Remarks

The **describe** method returns a structure containing information defined in the IDL specification of a **Contained** object. The specified object represents a component of an IDL interface (class) definition maintained within the Interface Repository.

When finished using the information in the returned **Description** structure, the client code must release the storage allocated for it. To free the associated storage, use a call similar to this:

```
if (desc.value._value)
     SOMFree (desc.value._value);
```

CAUTION:
The describe method returns pointers to elements within objects (for example, name). Thus, the somFree method should *not* be used to release any of these objects while the describe information is still needed.

describe

The "name" field of the **Description** is the name of the type of description. The "name" values are from the following set:

{"ModuleDescription", "InterfaceDescription", "AttributeDescription", "OperationDescription", "ParameterDescription", "TypeDescription", "ConstantDescription", "ExceptionDescription"}

The "value" field is a structure of type **any** whose "_value" field is a pointer to a structure of the type named by the "name" field of the **Description**. This structure provides all of the information contained in the IDL specification of the *receiver*. For example, if the **describe** method is invoked on an object of type **AttributeDef,** the "name" field of the returned **Description** will contain the identifier "AttributeDescription" and the "value" field will contain an **any** structure whose "_value" field is a pointer to an **AttributeDescription** structure.

Original Class

Contained

Related Methods

Methods

- **within**

Example Code

```
#include <containd.h>
#include <attribdf.h>
#include <somtc.h>

...
AttributeDef attr; /* An AttributeDef object (also a Contained) */
Description desc; /* .value field will be an AttributeDescription */
AttributeDescription *ad;
Environment *ev;

...
desc = Contained_describe (attr, ev);
ad = (AttributeDescription *) desc.value._value;
printf ("Attribute name: %s, defined in: %s\n",
         ad->name, ad->defined_in);
printf ("Attribute type: ");
TypeCode_print (ad->type, ev);
printf ("Attribute mode: %s\n", ad->mode == AttributeDef_READONLY ?
        "READONLY" : "NORMAL");
SOMFree (desc.value._value); /* Finished with describe output */
SOMObject_somFree (attr);    /* Finished with AttributeDef object */
```

within

This method returns a list of objects (in the Interface Repository) that contain a specified **Contained** object.

Syntax

```
sequence <container> within  (Contained  receiver, Environment  *env)
```

Parameters

receiver (Contained)
A pointer to a **Contained** object for which containing objects are needed.

env (Environment *)
A pointer to the **Environment** structure for the caller.

Returns

rc (sequence <container>)
Returns a sequence of **Container** objects that contain the specified **Contained** object.

Remarks

The **within** method returns a sequence of objects within the Interface Repository that contain the specified **Contained** object. If the receiving object is an **InterfaceDef** or **ModuleDef**, it can only be contained by the object that defines it. Other objects can be contained by objects that define or inherit them.

If the object is global in scope, the sequence returned by **within** will have its **_length** field set to zero.

When finished using the sequence returned by this method, the client code is responsible for releasing each of the **Container**s in the sequence and freeing the sequence buffer. In C, this can be accomplished as follows:

```
if (seq._length) {
    long i;
    for (i=0; i
         _somFree (seq._buffer[i]); /* Release each Container obj */
    SOMFree (seq._buffer);        /* Release the sequence buffer */
}
```

Original Class

Contained

Related Methods

Methods

- **describe**

Example Code

```
#include <containd.h>
#include <containr.h>

...
Contained anObj;
Environment *ev;
sequence(Container) sc;
long i;
...
sc = Contained_within (anObj, ev);
printf ("%s is contained in (or inherited by):\n",
     Contained__get_name (anObj, ev));
for (i=0; i
     printf ("\t%s\n",
         Contained__get_name ((Contained) sc._buffer[i], ev));
     SOMObject_somFree (sc._buffer[i]);
}
if (sc._length)
     SOMFree (sc._buffer);
```

Container

File stem: containr

Base

SOMObject

Metaclass

SOMClass

Ancestor Classes

SOMObject

Description

Types

```
typedef  string InterfaceName;
// Valid values for InterfaceName are limited to the following set:
//  {"AttributeDef", "ConstantDef", "ExceptionDef", "InterfaceDef",
//   "ModuleDef", "ParameterDef", "OperationDef", "TypeDef", "all"}

struct ContainerDescription  {
   Contained  * contained_object;
   Identifier  name;
   any   value;
};
```

New methods

The following list shows all the Container methods.

- contents
- lookup_name
- describe_contents

Overridden methods

The following list shows all the methods overridden by the Container class. These methods are overridden in order to modify the behavior defined by an ancestor class.

- somInit
- somUninit
- somDumpSelf
- somDumpSelfInt

contents

This method returns a sequence indicating the objects contained within a specified **Container** object of the Interface Repository.

Syntax

```
sequence<Contained> contents (Container  receiver, Environment  *env,
                              InterfaceName  limit_type,
                              boolean  exclude_inherited)
```

Parameters

receiver (Container)
A pointer to a **Container** object whose contained objects are needed.

env (Environment *)
A pointer to the **Environment** structure for the caller.

limit_type (InterfaceName)
The name of one interface type (see the valid list above) or "all", to specify what type of objects the **contents** method should search for.

exclude_inherited (boolean)
A **boolean** value: TRUE to exclude any inherited objects, or FALSE to include all objects.

Returns

rc (sequence<Contained>)
Returns a sequence of pointers to objects contained within the specified **Container** object.

Remarks

The **contents** method returns a list of objects contained by the specified **Container** object. Each object represents a component of an IDL interface (class) definition maintained within the Interface Repository.

The **contents** method is used to navigate through the hierarchy of objects within the Interface Repository. Starting with the **Repository** object, this method can list all of the objects in the Repository, then all of the objects within the **ModuleDef** objects, then all within the **InterfaceDef** objects, and so on.

contents

If the *"limit_type"* is set to "all", objects of all interface types are returned; otherwise, only objects of the requested interface type are returned. Valid values for **InterfaceName** are limited to the following set:

{"AttributeDef", "ConstantDef", "ExceptionDef", "InterfaceDef", "ModuleDef", "ParameterDef", "OperationDef", "TypeDef", "all"}

If *"exclude_inherited"* is set to TRUE, any inherited objects will *not* be returned.

When finished using the sequence returned by this method, the client code is responsible for releasing each of the objects in the sequence and freeing the sequence buffer. In C, this can be accomplished as follows:

```
if (seq._length) {
    long i;
    for (i=0; i <seq._length;i++)
       SOMObject_somFree (seq._buffer[i]); /* Release each object */
    SOMFree (seq._buffer);                 /* Release the buffer  */
}
```

Original Class

Container

Related Methods

Methods

- **lookup_name**
- **describe_contents**

Example Code

```
#include <containr.h>

...

Container anObj;
Environment *ev;
sequence(Contained) sc;
long i;

...

sc = Container_contents (anObj, ev, "all", TRUE);
printf ("%s contains the following objects:\n",
    SOMObject_somIsA (anObj, _Contained) ?
        Contained__get_name ((Contained) anObj, ev) :
        "The Interface Repository");
for (i=0; i <seq._length;i++) {
    printf ("\t%s\n",
        Contained__get_name (sc._buffer[i], ev));
    SOMObject_somFree (sc._buffer[i]);
}
if (sc._length)
    SOMFree (sc._buffer);
else
    printf ("\t[none]\n");
```

describe_contents

This method returns a sequence of descriptions of the objects contained within a specified **Container** object of the Interface Repository.

Syntax

```
sequence <ContainerDescription> describe_contents (Container  receiver,
                                                   Environment  *env,
                                                   InterfaceName  limit_type,
                                                   boolean  exclude_inherited,
                                                   long  max_returned_objs)
```

Parameters

receiver (Container)
A pointer to a **Container** object whose contained object descriptions are needed.

env (Environment *)
A pointer to the **Environment** structure for the caller.

limit_type (InterfaceName)
The name of one interface type (see the valid list above) or "all", to specify what type of objects the **describe_contents** method should return.

exclude_inherited (boolean)
A **boolean** value: TRUE to exclude any inherited objects, or FALSE to include all objects.

max_returned_objs (long)
A **long** integer indicating the maximum number of objects to be returned by the method, or -1 to indicate no limit is set.

Returns

rc (sequence <ContainerDescription>)
Returns a sequence of **ContainerDescription** structures, one for each object contained within the specified **Container** object. Each **ContainerDescription** structure has a "contained_object" field, which points to the contained object, as well as "name" and "value" fields, which are the result of the **describe** method.

Remarks

The **describe_contents** method combines the operations of the **contents** method and the **describe** method. That is, for each object returned by the **contents** operation, the description of the object is returned by invoking its **describe** operation. Each object represents a component of an IDL interface (class) definition maintained within the Interface Repository.

If the *"limit_type"* is set to "all", objects of all interface types are returned; otherwise, only objects of the requested interface type are returned. Valid values for **InterfaceName** are limited to the following set:

{"AttributeDef", "ConstantDef", "ExceptionDef", "InterfaceDef", "ModuleDef", "ParameterDef", "OperationDef", "TypeDef", "all"}

If *"exclude_inherited"* is set to TRUE, any inherited objects will *not* be returned.

The *"max_returned_objs"* argument is used to limit the number of objects that can be returned. If *"max_returned_objs"* is set to -1, the results for all contained objects will be returned.

When finished using the sequence returned by this method, the client code is responsible for freeing the "value._value" field in each description, releasing each of the objects in the sequence, and freeing the sequence buffer. In C, this can be accomplished as follows:

```
if (seq._length) {
    long i;
    for (i=0; i <seq._length;i++) {
        if (seq._buffer[i].value._value)
                                    /* Release each description */
            SOMFree (seq._buffer[i].value._value);
        SOMObject_somFree (seq._buffer[i].contained_object);
                                     /* Release each object */
    }
    SOMFree (seq._buffer);           /* Release the buffer  */
}
```

Original Class

Container

Related Methods

Methods

- **contents**
- **describe**
- **lookup_name**

describe_contents

Example Code

```
#include <containr.h>

...

Container anObj;
Environment *ev;
sequence(ContainerDescription) sc;
long i;

...

sc = Container_describe_contents (anObj, ev, "all", FALSE, -1L);
printf ("%s defines or inherits the following objects:\n",
    SOMObject_somIsA (anObj, _Contained) ?
        Contained__get_name ((Contained) anObj, ev) :
        "The Interface Repository");
for (i=0; i
     printf ("\t%s\n", sc._buffer[i].name);
     if (sc._buffer[i].value._value)
         SOMFree (sc._buffer[i].value._value);
     SOMObject_somFree (sc._buffer[i].contained_object);
}
if (sc._length)
    SOMFree (sc._buffer);
else
    printf ("\t[none]\n");
```

lookup_name

This method locates an object by name within a specified **Container** object of the Interface Repository, or within objects contained in the **Container** object.

Syntax

```
sequence<Contained> lookup_name (Container receiver,
                                 Environment *env, Identifier search_name,
                                 long levels_to_search, InterfaceName limit_type,
                                 boolean exclude_inherited)
```

Parameters

receiver (Container)
A pointer to a **Container** object in which to locate the object.

env (Environment *)
A pointer to the **Environment** structure for the caller.

search_name (Identifier)
The name of the object to be located.

levels_to_search (long)
A long having the value 1 or -1.

limit_type (InterfaceName)
The name of one interface type (see the valid list above) or “all”, to specify what type of object to search for.

exclude_inherited (boolean)
A **boolean** value: TRUE to exclude an object when it is inherited, or FALSE to return the object from wherever it is found.

Returns

rc (sequence<Contained>)
Returns a sequence of pointers to objects of the given name contained within the specified **Container** object, or within objects contained in the **Container** object.

lookup_name

Remarks

The **lookup_name** method locates an object by name within a specified **Container** object, or within objects contained in the **Container** object. The *"search_name"* specifies the name of the object to be found. Each object represents a component of an IDL interface (class) definition maintained within the Interface Repository.

The *"levels_to_search"* argument controls whether the lookup is constrained to the specified **Container** object or whether objects contained within the **Container** object are also searched. The *"levels_to_search"* value should be -1 to search the **Container** and all contained objects; it should be 1 to search only the **Container** itself.

If *"limit_type"* is set to "all", the lookup locates an object of the specified name with any interface type; otherwise, the search locates the object only if it has the designated interface type. Valid values for **InterfaceName** are limited to the following set:

{"AttributeDef", "ConstantDef", "ExceptionDef",
"InterfaceDef", "ModuleDef", "ParameterDef",
"OperationDef", "TypeDef", "all"}

If *"exclude_inherited"* is set to TRUE, any inherited objects will *not* be returned.

When finished using the sequence returned by this method, the client code is responsible for releasing each of the objects in the sequence and freeing the sequence buffer. In C, this can be accomplished as follows:

```
if (seq._length) {
    long i;
    for (i=0; i <seq._length;i++) {
        SOMObject_somFree (seq._buffer[i]);
                                        /* Release each object */
    SOMFree (seq._buffer);              /* Release the buffer  */
    }
```

Original Class

Container

Related Methods

Methods

- **contents**
- **describe_contents**

Example Code

```
#include <containr.h>
#include <containd.h>
#include <repostry.h>

...

Container repo;
Environment *ev;
sequence(Contained) sc;
long i;
Identifier nameToFind;

...

repo = (Container) RepositoryNew ();
sc = Container_lookup_name (repo, ev, nameToFind, -1, "all", TRUE);
printf ("%d object%s found:\n",
          sc._length, sc._length == 1 ? "" : "s");
for (i=0; i <seq._length;i++) {
    printf ("\t%s\n",
        Contained__get_id (sc._buffer[i], ev));
    SOMObject_somFree (sc._buffer[i]);
}
if (sc._length)
    SOMFree (sc._buffer);
```

ExceptionDef

File stem: excptdef

Base

Contained

Metaclass

SOMClass

Ancestor Classes

Contained
    SOMObject

Description

Types

```
struct ExceptionDescription  {
    Identifier      name;
    RepositoryId    id;
    RepositoryId    defined_in;
    TypeCode        type;
};
```

The **describe** method, inherited from **Contained,** returns an **ExceptionDescription** structure in the "value" member of the **Description** structure (defined in the **Contained** class).

Attributes

Listed below is each available attribute, with its corresponding type in parentheses, followed by a description of its purpose:

type (TypeCode)
    The **TypeCode** that represents the type of the **exception**. The **TypeCode** returned by the "_get_" form of the **type** attribute is contained in the receiving **ExceptionDef** object, which retains ownership. Hence, the returned **TypeCode** should not be freed. To obtain a separate copy, use the **TypeCode_copy** operation. The "_set_" form of the attribute makes a private copy of the **TypeCode** you supply, to keep in the receiving object. You retain ownership of the passed **TypeCode**.

New methods

There are currently no new methods defined for the ExceptionDef class.

Overridden methods

The following list shows all the methods overridden by the ExceptionDef class. These methods are overridden in order to modify the behavior defined by an ancestor class.

- somInit
- somUninit
- somDumpSelf
- somDumpSelfInt
- describe

InterfaceDef

File stem: intfacdf

Base

Contained
Container

Metaclass

SOMClass

Ancestor Classes

Contained
Container
SOMObject

Description

Types

```
struct FullInterfaceDescription   {
    Identifier    name;
    RepositoryId  id;
    RepositoryId  defined_in;
    sequence<OperationDef::OperationDescription> operation;
    sequence<AttributeDef::AttributeDescription> attributes;
};

struct InterfaceDescription   {
    Identifier    name;
    RepositoryId  id;
    RepositoryId  defined_in;
};
```

The **describe** method, inherited from **Contained,** returns an **InterfaceDescription** structure in the "value" member of the **Description** structure (defined in the **Contained** class). The **describe_contents** method, inherited from **Container,** returns a sequence of these **Description** structures, each carrying a reference to an **InterfaceDescription** structure in its "value" member.

Implementation note: The two sequences "OperationDescription" and "AttributeDescription" are built dynamically within the **FullInterfaceDescription** structure, due to the **InterfaceDef** class's inheritance from the **Contained** class.

Attributes

All attributes of the **InterfaceDef** class provide access to information kept within the receiving **InterfaceDef** object. The "_get_" form of the attribute returns a memory reference that is only valid as long as the receiving object has not been freed (using **_somFree**). The "_set_" form of the attribute makes a (deep) copy of your data and places it in the receiving **InterfaceDef** object. You retain ownership of all memory references passed using the "_set_" attribute forms.

Listed below is each available attribute, with its corresponding type in parentheses, followed by a description of its purpose:

base_interfaces (sequence <RepositoryId>)
The sequence of **RepositoryId**s for all of the interfaces that the receiving interface inherits.

instanceData (TypeCode)
The **TypeCode** of a structure whose members are the internal instance variables, if any, described in the SOM **implementation** section of the interface.

Note: This attribute is a SOM-unique extension of the Interface Repository; it is not stipulated by the CORBA specifications.

New methods

The following list shows all the InterfaceDef methods.

- describe_interface

Overridden methods

The following list shows all the methods overridden by the InterfaceDef class. These methods are overridden in order to modify the behavior defined by an ancestor class.

- somInit
- somUninit
- somDumpSelf
- somDumpSelfInt
- within
- describe

describe_interface

This method returns (from the Interface Repository) a description of all the methods and attributes of an interface definition.

Syntax

```
FullInterfaceDescription describe_interface (InterfaceDef receiver,
                                             Environment *env)
```

Parameters

receiver (InterfaceDef)
A pointer to an object of class **InterfaceDef** representing the Interface Repository object where an interface definition is stored.

env (Environment *)
A pointer where the method can return exception information if an error is encountered.

Returns

rc (FullInterfaceDescription)
Returns a description of all the methods and attributes of an interface definition that are held in the Interface Repository.

Remarks

The **describe_interface** method returns a description of all the methods and attributes of an interface definition that are held in the Interface Repository.

When finished using the **FullInterfaceDescription** returned by this method, the client code is responsible for freeing the _buffer fields of the two sequences it contains. In C, this can be accomplished as follows:

```
if (fid.operation._length)
    SOMFree (fid.operation._buffer);        /* Release the buffer */
if (fid.attributes._length)
    SOMFree (fid.attributes._buffer);       /* Release the buffer */
```

Original Class

InterfaceDef

Example Code

```
#include <intfacdf.h>

...

InterfaceDef idef;
Environment *ev;
FullInterfaceDescription fid;
long i;

...
fid = InterfaceDef_describe_interface (idef, ev);
printf ("The %s interface has the following
attributes:\n",
    Contained__get_name ((Contained) idef, ev));
if (!fid.attributes._length)
    printf ("\t[none]\n");
else {
    for (i=0; i <fid.attributes._length;i++) {
        printf ("\t%s\n",
fid.attributes._buffer[i].name);
    SOMFree (fid.attributes._buffer);
}
printf ("and the following methods:\n")
if (!fid.operation._length)
    printf ("\t[none]\n");
else {
    for (i=0; i <fid.attributes._length;i++) {
        printf ("\t%s\n", fid.operation._buffer[i].name);
    SOMFree (fid.operation._buffer);
}
```

ModuleDef

File stem: moduledf

Base

Contained
Container

Metaclass

SOMClass

Ancestor Classes

Contained
Container
SOMObject

Description

Types

```
struct ModuleDescription {
    Identifier    name;
    RepositoryId  id;
    RepositoryId  defined_in;
};
```

The **describe** method, inherited from **Contained**, returns a **ModuleDescription** structure in the "value" member of the **Description** structure (defined in the **Contained** class). The **describe_contents** method, inherited from **Container,** returns a sequence of these **Description** structures, each carrying a reference to a **ModuleDescription** structure in its "value" member.

New methods

There are currently no new methods defined for the ModuleDef class.

Overridden methods

The following list shows all the methods overridden by the ModuleDef class. These methods are overridden in order to modify the behavior defined by an ancestor class.

- somInit
- somUninit
- somDumpSelf
- somDumpSelfInt
- describe
- within

OperationDef

File stem: operatdf

Base

Contained
Container

Metaclass

SOMClass

Ancestor Classes

Contained
    Container
        SOMObject

Description

Types

```
typedef  Identifier  ContextIdentifier;
enum  OperationMode   { NORMAL, ONEWAY };

struct  OperationDescription   {
    Identifier      name;
    RepositoryId    id;
    RepositoryId    defined_in;
    TypeCode        result;
    OperationMode   mode;
    sequence <ContextIdentifier>  contexts;
    sequence <ParameterDef::ParameterDescription>  parameter;
    sequence < ExceptionDef::ExceptionDescription>  exceptions;
};
```

The **describe** method, inherited from **Contained,** returns an **OperationDescription** structure in the "value" member of the **Description** structure (defined in the **Contained** class). The **describe_contents** method, inherited from **Container,** returns a sequence of these **Description** structures, each carrying a reference to an **OperationDescription** structure in its "value" member.

Attributes

Listed below is each available attribute, with its corresponding type in parentheses, followed by a description of its purpose:

result (TypeCode)
The **TypeCode** that represents the type of the operation (method). The **TypeCode** returned by the "_get_" form of the **type** attribute is contained in the receiving **OperationDef** object, which retains ownership. Hence, the returned **TypeCode** should not be freed. To obtain a separate copy, use the **TypeCode_copy** operation. The "_set_" form of the attribute makes a private copy of the **TypeCode** you supply, to keep in the receiving object. You retain ownership of the passed **TypeCode**.

mode (OperationMode)
The **OperationMode** of the operation (method), either NORMAL or ONEWAY.

contexts (sequence <ContextIdentifier>)
The list of **ContextIdentifier**s associated with the operation (method). The "_get_" form of the attribute returns a sequence whose buffer is owned by the receiving **OperationDef** object. You should not free it. The "_set_" form of the attribute makes a (deep) copy of the passed sequence; you retain ownership of the original storage.

New methods

There are currently no new methods defined for the OperationDef class.

Overridden methods

The following list shows all the methods overridden by the OperationDef class. These methods are overridden in order to modify the behavior defined by an ancestor class.

- somInit
- somUninit
- somDumpSelf
- somDumpSelfInt
- describe

ParameterDef

File stem: paramdef

Base

Contained

Metaclass

SOMClass

Ancestor Classes

Contained
    SOMObject

Description

Types

```
enum  ParameterMode  { IN, OUT, INOUT };

struct ParameterDescription {
    Identifier      name;
    RepositoryId    id;
    RepositoryId    defined_in;
    TypeCode        type;
    ParameterMode   mode;
};
```

The **describe** method, inherited from **Contained,** returns a **ParameterDescription** structure in the "value" member of the **Description** structure (defined in the **Contained** class).

Attributes

Listed below is each available attribute, with its corresponding type in parentheses, followed by a description of its purpose:

type (TypeCode)
    The **TypeCode** that represents the type of the **parameter**. The **TypeCode** returned by the "_get_" form of the **type** attribute is contained in the receiving **ParameterDef** object, which retains ownership. Hence, the returned **TypeCode** should not be freed. To obtain a separate copy, use the **TypeCode_copy** operation. The "_set_" form of the attribute makes a private copy of the **TypeCode** you supply, to keep in the receiving object. You retain ownership of the passed **TypeCode**.

mode (ParameterMode)
The **ParameterMode** of the **parameter** (IN, OUT, or INOUT).

New methods

There are currently no new methods defined for the ParameterDef class.

Overridden methods

The following list shows all the methods overridden by the ParameterDef class. These methods are overridden in order to modify the behavior defined by an ancestor class.

- somInit
- somUninit
- somDumpSelf
- somDumpSelfInt
- describe

Repository

File stem: repostry

Base

Container

Metaclass

SOMClass

Ancestor Classes

Container

SOMObject

Description

Types

```
struct RepositoryDescription  {
    Identifier  name;
    RepositoryId  id;
    RepositoryId  defined_in;
};
```

The inherited **describe_contents** method returns an instance of the **RepositoryDescription** structure in the "value" member of the **Description** structure defined in the **Container** interface.

New methods

The following list shows all the Repository methods.

- lookup_id
- lookup_modifier
- release_cache

Overridden methods

The following list shows all the methods overridden by the Repository class. These methods are overridden in order to modify the behavior defined by an ancestor class.

- describe_contents
- somInit
- somUninit
- somFree
- somDumpSelf
- somDumpSelfInt

lookup_id

This method returns the object having a specified **RepositoryId**.

Syntax

```
Contained lookup_id  (Repository  receiver, Environment  *env,
                      RepositoryId  search_id)
```

Parameters

receiver (Repository)
A pointer to an object of class **Repository** representing SOM's Interface Repository.

env (Environment *)
A pointer where the method can return exception information if an error is encountered.

search_id (RepositoryId)
An ID value of type **RepositoryId** that uniquely identifies the desired object in the Interface Repository.

Returns

rc (Contained)
Returns the **Contained** object that has the specified **RepositoryId**.

Remarks

The **lookup_id** method returns the object having a **RepositoryId** given by the specified *search_id* argument. The returned object represents a component of an IDL interface (class) definition maintained within the Interface Repository.

When finished using the object returned by this method, the client code is responsible for releasing it, using the **somFree** method.

Original Class

Repository

Related Methods

Methods

- **lookup_modifier**
- **lookup_name**
- **contents**
- **within**

Example Code

```
#include <containd.h>
#include <repostry.h>

...

Repository repo;
Environment *ev;
Contained c;
RepositoryId objectToFind;

...

repo = RepositoryNew ();
c = Repository_lookup_id (repo, ev, objectToFind);
if (c) {
    printf ("lookup_id found object of type: %s, named: %s\n",
        SOMObject_somGetClassName (c), Contained__get_name (c, ev));
    SOMObject_somFree (c);
}
```

lookup_modifier

Returns the value of a given SOM **modifier** for a specified object [that is, for an object that is a component of an IDL interface (class) definition maintained within the Interface Repository].

Syntax

```
string lookup_modifier  (Repository  receiver, Environment  *env,
                         RepositoryId  id, string  modifier)
```

Parameters

receiver (Repository)
A pointer to an object of class **Repository** representing SOM's Interface Repository.

env (Environment *)
A pointer where the method can return exception information if an error is encountered.

id (RepositoryId)
The **RepositoryId** of the object whose modifier value is needed.

modifier (string)
The name of a specific (SOM or user-specified) modifier whose string value is needed.

Returns

rc (string)
The **lookup_modifier** method returns the string value of the given SOM **modifier** for an object with the specified **RepositoryId**, if it exists. If an existing modifier has no value, a zero-length string value is returned. If the object cannot be found, then NULL (or zero) is returned.

When the string value is no longer needed, client code must free the space for the string (using **SOMFree**).

Remarks

The **lookup_modifier** method returns the string value of the given SOM **modifier** for an object with the specified **RepositoryId** within the Interface Repository. For a

discussion of SOM modifiers, see the topic "Modifier statements" in Chapter 4, "SOM IDL and The SOM Compiler," of the *SOM Programming Guide*.

If the object with the given **RepositoryId** does not exist or does not possess the modifier, then NULL (or zero) is returned. If the object exists but the specified modifier does not have a value, a zero-length string value is returned.

Note: The **lookup_modifier** method is *not* stipulated by the CORBA specifications; it is a SOM-unique extension to the Interface Repository.

Original Class

Repository

Related Methods

Methods

- **lookup_id**
- **lookup_name**

Example Code

```
#include <repostry.h>

...

Repository repo;
Environment *ev;
RepositoryId objectId;
string filestem;i

...

repo = RepositoryNew ();
filestem = Repository_lookup_modifier (repo, ev, objectId,
                                                    "filestem");
if (filestem) {
    printf
        ("The %s object's filestem modifier has the value \"%s\"\n",
            objectId, filestem);
    SOMFree (filestem);
} else
    printf ("No filestem modifier could be found for %s\n",
            objectId);
```

release_cache

This method permits the Repository object to release the memory occupied by Interface Repository objects that have been implicitly referenced.

Syntax

```
void release_cache   (Repository  receiver, Environment  *env)
```

Parameters

receiver (Repository)
A pointer to an object of class **Repository** representing SOM's Interface Repository.

env (Environment *)
A pointer where the method can return exception information if an error is encountered.

Returns

rc (void)

Remarks

This method allows the Repository object to release the memory occupied by implicitly referenced Interface Repository objects. Some methods (such as **describe_contents** and **lookup_name**) may cause some objects to be instantiated that are not directly accessible through object references that have been returned to the user. These objects are kept in an internal Interface Repository cache until the **release_cache** method is used to free them. The internal cache continuously replenishes itself over time as the need arises.

See the section entitled “A word about memory management” in the Interface Repository Framework in the *SOM Programming Guide*.

Original Class

Repository

Example Code

```
#include <repostry.h>

...

Repository repo;
Environment *ev;
sequence(ContainerDescription) scd;

...

scd = Container_describe_contents (
      (Container) repo, ev, "TypeDef", TRUE, -1);
Repository_release_cache (repo, ev);
```

TypeDef

File stem: typedef

Base

Contained

Metaclass

SOMClass

Ancestor Classes

Contained

SOMObject

Description

Types

```
struct      TypeDescription         {
Identifier    name;
RepositoryId  id;
RepositoryId  defined_in;
TypeCode      type;
};
```

The **describe** method, inherited from **Contained,** returns a **TypeDescription** structure in the "value" member of the **Description** structure (defined in the **Contained** class).

Attributes

Listed below is each available attribute, with its corresponding type in parentheses, followed by a description of its purpose:

type (TypeCode)

The **TypeCode** that represents the type of the typedef. The **TypeCode** returned by the "_get_" form of the **type** attribute is contained in the receiving **TypeDef** object, which retains ownership. Hence, the returned **TypeCode** should not be freed. To obtain a separate copy, use the **TypeCode_copy** operation. The "_set_" form of the attribute makes a private copy of the **TypeCode** you supply, to keep in the receiving object. You retain ownership of the passed **TypeCode**.

New methods

There are currently no new methods defined for the TypeDef class.

Overridden methods

The following list shows all the methods overridden by the TypeDef class. These methods are overridden in order to modify the behavior defined by an ancestor class.

- somInit
- somUninit
- somDumpSelf
- somDumpSelfInt
- describe

TypeCode_alignment

This function supplies the alignment value for a given **TypeCode**.

Syntax

```
short TypeCode_alignment  (TypeCode  tc, Environment  *env)
```

Parameters

tc (TypeCode)
The **TypeCode** whose alignment information is desired.

env (Environment *)
A pointer to an Environment structure.

Returns

rc (short)
A short integer containing the alignment value.

Remarks

This function returns the alignment information associated with the given **TypeCode**. The alignment value is a short integer that should evenly divide any memory address where an instance of the type described by the **TypeCode** will occur.

Related Information

Functions

- **TypeCodeNew**
- **TypeCode_equal**
- **TypeCode_kind**
- **TypeCode_param_count**
- **TypeCode_parameter**
- **TypeCode_setAlignment**
- **TypeCode_size**
- **TypeCode_free**
- **TypeCode_print**

TypeCode_copy

This function creates a new copy of a given **TypeCode**.

Syntax

```
TypeCode TypeCode_copy   (TypeCode  tc, Environment  *env)
```

Parameters

tc (TypeCode)
The **TypeCode** to be copied.

env (Environment *)
A pointer to an **Environment** structure. The CORBA standard mandates the use of this structure as a standard way to return exception information when an error condition is detected.

Returns

rc (TypeCode)
A new **TypeCode** with no internal references to any previously existing **TypeCode**s or **string**s. If a copy cannot be created successfully, the value NULL is returned. No exceptions are raised by this function.

Remarks

The **TypeCode_copy** function creates a new copy of a given **TypeCode**. **TypeCode**s are complex data structures whose actual representation is hidden and may contain internal references to **string**s and other **TypeCode**s. The copy created by this function is guaranteed not to refer to any previously existing **TypeCode**s or **string**s, and hence can be used long after the original **TypeCode** is freed or released (**TypeCode**s are typically contained in Interface Repository objects whose memory resources are released by the **_somFree** method).

All of the memory used to construct the **TypeCode** copy is allocated dynamically and should be subsequently freed *only* by using the **TypeCode_free** function.

This function is a SOM-unique extension to the CORBA standard.

TypeCode_copy

Related Information

Functions

- **TypeCodeNew**
- **TypeCode_alignment**
- **TypeCode_equal**
- **TypeCode_kind**
- **TypeCode_param_count**
- **TypeCode_parameter**
- **TypeCode_size**
- **TypeCode_free**
- **TypeCode_print**
- **TypeCode_setAlignment**

TypeCode_equal

This function compares two **TypeCode**s for equality.

Syntax

```
boolean TypeCode_equal  (TypeCode  tc, Environment  *env,
                         TypeCode  tc2)
```

Parameters

tc (TypeCode)
One of the **TypeCode**s to be compared.

env (Environment *)
A pointer to an **Environment** structure. The CORBA standard mandates the use of this structure as a standard way to return exception information when an error condition is detected.

tc2 (TypeCode)
The other **TypeCode** to be compared.

Returns

rc (boolean)
Returns TRUE (1) if the **TypeCode**s *tc* and *tc2* describe the same data type, with the same alignment. Otherwise, FALSE (0) is returned. No exceptions are raised by this function.

Remarks

The **TypeCode_equal** function can be used to determine if two distinct **TypeCode**s describe the same underlying abstract data type.

Related Information

Functions

- **TypeCodeNew**
- **TypeCode_alignment**
- **TypeCode_kind**
- **TypeCode_param_count**
- **TypeCode_parameter**
- **TypeCode_copy**

TypeCode_equal

- **TypeCode_free**
- **TypeCode_print**
- **TypeCode_setAlignment**
- **TypeCode_size**

TypeCode_free

This function destroys a given **TypeCode** by freeing all of the memory used to represent it.

Syntax

```
void TypeCode_free  (TypeCode  tc, Environment  *env)
```

Parameters

tc (TypeCode)
The **TypeCode** to be freed.

env (Environment *)
A pointer to an **Environment** structure. The CORBA standard mandates the use of this structure as a standard way to return exception information when an error condition is detected.

Returns

rc (void)

Remarks

The **TypeCode_free** function destroys a given **TypeCode** by freeing all of the memory used to represent it. **TypeCode**s obtained from the **TypeCode_copy** or **TypeCodeNew** functions should be freed using **TypeCode_free**. **TypeCode**s contained in Interface Repository objects should never be freed. Their memory is released when a **_somFree** method releases the Interface Repository object.

The **TypeCode_free** operation has no effect on **TypeCode** constants. **TypeCode** constants are static **TypeCode**s declared in the header file "somtcnst.h" or generated in files emitted by the SOM Compiler. Since **TypeCode** constants may be used interchangeably with dynamically created **TypeCode**s, it is *not* considered an error to attempt to free a **TypeCode** constant with the **TypeCode_free** function.

This function is a SOM-unique extension to the CORBA standard.

TypeCode_free

Related Information

Functions

- **TypeCodeNew**
- **TypeCode_alignment**
- **TypeCode_equal**
- **TypeCode_kind**
- **TypeCode_param_count**
- **TypeCode_parameter**
- **TypeCode_size**
- **TypeCode_copy**
- **TypeCode_print**
- **TypeCode_setAlignment**

TypeCode_kind

This function categorizes the abstract data type described by a **TypeCode**.

Syntax

```
TCKind TypeCode_kind  (TypeCode  tc, Environment  *env)
```

Parameters

tc (TypeCode)
The **TypeCode** whose **TCKind** categorization is requested.

env (Environment *)
A pointer to an **Environment** structure.

The CORBA standard mandates the use of this structure as a standard way to return exception information when an error condition is detected.

Returns

rc (TCKind)
Returns one of the enumerators listed in the **TCKind** enumeration shown below. No exceptions are raised by this function. enum TCKind { tk_null, tk_void, tk_short, tk_long, tk_ushort,
tk_ulong, tk_float, tk_double, tk_boolean, tk_char,
tk_octet, tk_any, tk_TypeCode, tk_Principal,
tk_objref, tk_struct, tk_union, tk_enum, tk_string,
tk_sequence, tk_array, tk_pointer, tk_self,
tk_foreign
};

Remarks

The **TypeCode_kind** function can be used to classify a **TypeCode** into one of the categories listed in the **TCKind** enumeration. Based on the "kind" classification, a **TypeCode** may contain 0 or more additional parameters to fully describe the underlying data type.

Figure 4 on page 569 indicates the number and function of these additional parameters. **TCKind** entries not listed in the table are basic data types and do not

have any additional parameters. The designation “N” refers to the number of members in a **struct** or **union**, or the number of enumerators in an **enum**.

Figure 4 (Page 1 of 2). TypeCode information per TCKind category

TCKind	Parameters	Type	Function
tk_objref	1	**string**	The ID of the corresponding **InterfaceDef** in the Interface Repository
tk_struct	2N+1	**string**	The name of the **struct**.
		----- next 2 repeat for each member -----	
		string **TypeCode**	The name of the **struct** member. The type of the **struct** member.
tk_union	3N+2	**string** **TypeCode**	The name of the **union**. The type of the discriminator.
		----- next 3 repeat for each member -----	
		long **string** **TypeCode**	The label value The name of the member. The type of the member.

* The **TCKind** values **tk_pointer**, **tk_self**, and **tk_foreign** are SOM-unique extensions to the CORBA standard. They are provided to permit **TypeCodes** to describe types that cannot be expressed in standard IDL.

The **tk_pointer TypeCode** contains only one parameter—a **TypeCode** which describes the data type that the pointer references. The **tk_self TypeCode** is used to describe a "self-referential" structure or union without introducing unbounded recursion in the **TypeCode**. For example, the following C struct:

```
struct node {
   long count;
   struct node *next;
   };
```

could be described with a **TypeCode** created as follows:

```
TypeCode tcForNode;

tcForNode = TypeCodeNew (tk_struct, "node",
    "count", TypeCodeNew (tk_long),
    "next", TypeCodeNew (tk_pointer,
    TypeCodeNew (tk_self, "node")));
```

The **tk_foreign TypeCode** provides a more general escape mechanism, allowing **TypeCode**s to be created that partially describe non-IDL types. Since these foreign **TypeCode**s carry only a partial description of a type, the "implementation context" parameter can be used by a non-IDL execution environment to recognize other types that are known or understood in that environment. See the section entitled "Using the tk_foreign TypeCode" in Chapter 7 of the *SOM Programming Guide* for more information about using foreign **TypeCodes** in SOM IDL files.

Note that the use of self-referential structures, pointers, or foreign types is beyond the scope of the CORBA standard, and may result in a loss of portability or distributability in client code.

TypeCode_kind

Figure 4 (Page 2 of 2). TypeCode information per TCKind category

TCKind	Parameters	Type	Function
tk_enum	N+1	**string**	The name of the **enum**.
		--- next repeats for each enumerator ----	
		string	The name of the enumerator.
tk_string	1	**long**	The maximum string length or 0.
tk_sequence	2	**TypeCode** **long**	The type of element in the sequence. The maximum number of elements or 0.
tk_array	2	**TypeCode** **long**	The type of element in the **array**. The maximum number of elements.
tk_pointer*	1	**TypeCode**	The type of the referenced datum.
tk_self*	1	**string**	The name of the referenced enclosing **struct** or **union**.
tk_foreign*	3	**string** **string** **long**	The name of the foreign type. The implementation context. The size of an instance.

* The **TCKind** values **tk_pointer**, **tk_self**, and **tk_foreign** are SOM-unique extensions to the CORBA standard. They are provided to permit **TypeCodes** to describe types that cannot be expressed in standard IDL.

The **tk_pointer TypeCode** contains only one parameter—a **TypeCode** which describes the data type that the pointer references. The **tk_self TypeCode** is used to describe a "self-referential" structure or union without introducing unbounded recursion in the **TypeCode**. For example, the following C struct:

```
struct node {
   long count;
   struct node *next;
   };
```

could be described with a **TypeCode** created as follows:

```
TypeCode tcForNode;

tcForNode = TypeCodeNew (tk_struct, "node",
    "count", TypeCodeNew (tk_long),
    "next", TypeCodeNew (tk_pointer,
    TypeCodeNew (tk_self, "node")));
```

The **tk_foreign TypeCode** provides a more general escape mechanism, allowing **TypeCode**s to be created that partially describe non-IDL types. Since these foreign **TypeCode**s carry only a partial description of a type, the "implementation context" parameter can be used by a non-IDL execution environment to recognize other types that are known or understood in that environment. See the section entitled "Using the tk_foreign TypeCode" in Chapter 7 of the *SOM Programming Guide* for more information about using foreign **TypeCodes** in SOM IDL files.

Note that the use of self-referential structures, pointers, or foreign types is beyond the scope of the CORBA standard, and may result in a loss of portability or distributability in client code.

Original Class

TypeDef

Related Information

Functions:

- **TypeCodeNew**
- **TypeCode_alignment**
- **TypeCode_equal**
- **TypeCode_param_count**
- **TypeCode_parameter**
- **TypeCode_copy**
- **TypeCode_free**
- **TypeCode_print**
- **TypeCode_setAlignment**
- **TypeCode_size**

TypeCodeNew

This function creates a new **TypeCode** instance.

Syntax

```
TypeCode TypeCodeNew  (TCKind tag, string interfaceID, string name,
                       string mbrName, string EnumId,
                       string structOrUnionName, long maxLength,
                       long length, long flag, long labelValue,
                       TypeCode mbrTC, TypeCode swTC,
                       TypeCode seqTC, TypeCode arrayTC,
                       TypeCode ptrTC, string typename,
                       string impCtx, long instSize,
                       TCKind allOtherTagValues)
```

Parameters

tag (TCKind)
The type or category of **TypeCode** to create. The actual parameters that follow are variable in number and type, depending on the value of the tag paramater. (There are no implicit parameters in this function.) See the remarks section for the syntax form appropriate for each kind of TCKind *tag*.

interfaceID (string)
A string containing the fully-qualified interface name that is the subject of an object reference type.

name (string)
A string that gives the name of a **struct, union,** or **enum**.

mbrName (string)
A string that gives the name of a **struct** or **union** member element.

EnumId (string)
A string that gives the name of an **enum** enumerator.

structOrUnionName (string)
A string that gives the name of a s**truct** or **union** that has been previously named in the current **TypeCode** and is the subject of a self-referential pointer type. See the footnote on **tk_self** in Table 1 for an example of what this means and how it is applied.

maxLength (long)
The maximum permitted length of a **string** or a **sequence**. The value 0 (zero) means that the **string** or **sequence** is considered unbounded.

length (long)
The maximum number of elements that can be stored in an array. All IDL arrays are bounded, hence a value of zero denotes an array of zero elements.

flag (long)
One of the following constant values used to distinguish a labeled case in an IDL discriminated **union** switch statement from the default case:

TCREGULAR_CASE - The value 1
TCDEFAULT_CASE - The value 2

labelValue (long)
The actual value associated with a regular labeled case in an IDL discriminated **union** switch statement. If preceded by the argument TCDEFAULT_CASE, the value zero should be used.

mbrTC (TypeCode)
A **TypeCode** that represents the data type of a **struct** or **union** member.

swTC (TypeCode)
A **TypeCode** that represents the data type of the discriminator in an IDL **union** statement.

seqTC (TypeCode)
A **TypeCode** that describes the data type of the elements in a **sequence**.

arrayTC (TypeCode)
A **TypeCode** that describes the data type of the elements of an **array**.

ptrTC (TypeCode)
A **TypeCode** that describes the data type referenced by a pointer.

typename (string)
A string that provides the name of a foreign type.

impCtx (string)
A string that identifies an implementation context where a foreign type is understood.

instSize (long)
A long that holds the size of a foreign type instance. If the size is variable or is not known, the value zero should be used.

TypeCodeNew

allOtherTagValues (TCKind)
One of the values: **tk_null, tk_void, tk_short, tk_long, tk_ushort, tk_ulong, tk_float, tk_double, tk_boolean, tk_char, tk_octet, tk_any, tk_TypeCode,** or **Tk_Principal**. All of these tags represent basic IDL data types that do not require any other descriptive parameters.

Returns

rc (TypeCode)
A new **TypeCode** instance, or NULL if the new instance could not be created.

Remarks

The **TypeCodeNew** function creates a new instance of a **TypeCode** from the supplied parameters. **TypeCode**s are complex data structures whose actual representation is hidden. The number and types of arguments required by **TypeCodeNew** varies depending on the value of the first argument. The syntax for all of the valid invocation sequences are as follows:

TyepCodeNew(tk_objref, string interfaceId);
TypeCodeNew(tk_string, long maxLength);
TypeCodeNew(tk_sequence, TypeCode seqTC, **long** maxLength);
TypeCodeNew(tk_array, TypeCodearrayTC, **long** length);
TypeCodeNew(tk_pointer, TypeCode ptrTC);
TypeCodeNew(tk_self, string structOrUnionName);
TypeCodeNew(tk_foreign, string typename, **string** impCtx, **long** instSize);
TypeCodeNew(tk_struct, string name, **string** mbrname,
TypeCode mbrTC,[...,]
[mbrName and mbrTC repeat as needed]
NULL);
TypeCodeNew (tk_union, string name, **TypeCode** swTC,
longflag, **long** labelValue, **string**
mbrName, **TypeCode** mbrTC,[...,]
[flag, labelValue, mbrName and mbrTC repeat as needed]
NULL);
TypeCodeNew(tk_enum, string name, **string** enumld,[...,]
[enumlds repeat as needed]
NULL);
TypeCodeNew(TCKindall OtherTAgValues);

All **TypeCode**s created by **TypeCodeNew** should be destroyed (when no longer needed) using the **TypeCode_free** function.

This function is a SOM-unique extension to the CORBA standard.

Related Information

Functions

- **TypeCode_alignment**
- **TypeCode_copy**
- **TypeCode_equal**
- **TypeCode_free**
- **TypeCode_kind**
- **TypeCode_param_count**
- **TypeCode_parameter**
- **TypeCode_print**
- **TypeCode_size**
- **TypeCode_setAlignment**

TypeCode_param_count

This function obtains the number of parameters available in a given **TypeCode**.

Syntax

```
long TypeCode_param_count  (TypeCode  tc, Environment  *env)
```

Parameters

tc (TypeCode)
The **TypeCode** whose parameter count is desired.

env (Environment *)
A pointer to an **Environment** structure. The CORBA standard mandates the use of this structure as a standard way to return exception information when an error condition is detected.

Returns

rc (long)
Returns the actual number of parameters associated with the given **TypeCode**, in accordance with Table 1. No exceptions are raised by this function.

Remarks

The **TypeCode_param_count** function can be used to obtain the actual number of parameters contained in a specified **TypeCode**. Each **TypeCode** contains sufficient parameters to fully describe its underlying abstract data type. Refer to Table 1.

Related Information

Functions

- **TypeCodeNew**
- **TypeCode_alignment**
- **TypeCode_equal**
- **TypeCode_kind**
- **TypeCode_parameter**
- **TypeCode_copy**
- **TypeCode_free**
- **TypeCode_print**
- **TypeCode_size**
- **TypeCode_setAlignment**

TypeCode_parameter

This function obtains a specified parameter from a given **TypeCode**.

Syntax

```
any TypeCode_parameter  (TypeCode tc, Environment *env, long index)
```

Parameters

tc (TypeCode)
The **TypeCode** whose parameter is desired.

env (Environment *)
A pointer to an **Environment** structure. The CORBA standard mandates the use of this structure as a standard way to return exception information when an error condition is detected.

index (long)
The number of the desired parameter. Parameters are numbered from 0 to N-1, where N is the value returned by the **Typecode_param_count** function.

Returns

rc (any)
Returns the requested parameter in the form of an **any**. This function raises the Bounds exception if the value of the index exceeds the number of parameters available in the given **TypeCode**. Because the values exist within the specified **TypeCode**, you should not free the results returned from this function.

A bounds exception is also raised if this function is called on any of the IDL basic data types (see note above).

An **any** is a basic IDL data type that is represented as the following structure in C or C++:

```
typedef struct any {
    TypeCode _type;
    void *   _value;
} any;
```

Since all **TypeCode** parameters have one of only three types (**string**, **TypeCode**, or **long**), the **_type** member will always be set to **TC_string**, **TC_TypeCode**, or **TC_long,** as appropriate. The **_value** member always points to the actual

parameter datum. For example, the following code can be used to extract the name of a structure from a **TypeCode** of kind **tk_struct** in C:

```
#include <repostry.h>  /* Interface Repository class */
#include <typedef.h>   /* Interface Repository TypeDef class */
#include <somtcnst.h>  /* TypeCode constants */
TypeCode x;
Environment *ev = somGetGlobalEnvironment ();
TypeDef aTypeDefObj;
sequence(Contained) sc;
any parm;
string name;
Repository repo;

...

/* 1st, obtain a TypeCode from an Interface Repository object,
 * or use a TypeCode constant.
 */

repo = RepositoryNew ();
sc = _lookup_name (repo, ev,
     "AttributeDescription", -1, "TypeDef", TRUE);
if (sc._length) {
    aTypeDefObj = sc._buffer[0];
    x = __get_type (aTypeDefObj, ev);
    }
else
    x = TC_AttributeDescription;

if (TypeCode_kind (x, ev) == tk_struct) {
    parm = TypeCode_parameter (x, ev, 0); /* Get structure name */
    if (TypeCode_kind (parm._type, ev) != tk_string) {
        printf ("Error, unexpected TypeCode: ");
        TypeCode_print (parm._type, ev);
    } else {
        name = *((string *)parm._value);
        printf ("The struct name is %s\n", name);
    }
} else {
    printf ("TypeCode is not a tk_struct: ");
    TypeCode_print (x, ev);
}
```

Remarks

The **TypeCode_parameter** function can be used to obtain any of the parameters contained in a given **TypeCode**. Refer to Table 1 for a list of the number and type of parameters associated with each category of **TypeCode**.

Note: This function should *not* be used for any of the IDL basic data types (that is, any types in the **TCKind** enumeration that are *not* listed in table 1 under the **TypeCode_Kind** function).

Related Information

Functions

- **TypeCodeNew**
- **TypeCode_alignment**
- **TypeCode_equal**
- **TypeCode_kind**
- **TypeCode_param_count**
- **TypeCode_copy**
- **TypeCode_free**
- **TypeCode_print**
- **TypeCode_size**
- **TypeCode_setAlignment**

TypeCode_print

This function writes all of the information contained in a given **TypeCode** to "stdout".

Syntax

```
void TypeCode_print  (TypeCode  tc, Environment  *env)
```

Parameters

tc (TypeCode)
The **TypeCode** to be examined.

env (Environment *)
A pointer to an **Environment** structure. The CORBA standard mandates the use of this structure as a standard way to return exception information when an error condition is detected.

Returns

rc (void)

Remarks

The **TypeCode_print** function can be used during program debugging to inspect the contents of a **TypeCode**. It prints (in a human-readable format) all of the information contained in the **TypeCode**. The format of the information shown by **TypeCode_print** is the same form that could be used by a C programmer to code the corresponding **TypeCodeNew** function call to create the **TypeCode**.

This function is a SOM-unique extension to the CORBA standard.

Related Information

Functions

- **TypeCodeNew**
- **TypeCode_alignment**
- **TypeCode_equal**
- **TypeCode_kind**
- **TypeCode_param_count**

TypeCode_print

- **TypeCode_parameter**
- **TypeCode_copy**
- **TypeCode_free**
- **TypeCode_size**
- **TypeCode_setAlignment**

TypeCode_setAlignment

This function sets the alignment value for a given **TypeCode**.

Syntax

```
void TypeCode_setAlignment  (TypeCode  tc, Environment  *env,
                             short  alignment)
```

Parameters

tc (TypeCode)
The **TypeCode** to receive the new alignment value.

env (Environment *)
A pointer to an Environment structure.

alignment (short)
A short integer that specifies the alignment value.

Returns

rc (void)

Related Information

Functions

- **TypeCodeNew**
- **TypeCode_alignment**
- **TypeCode_equal**
- **TypeCode_kind**
- **TypeCode_param_count**
- **TypeCode_parameter**
- **TypeCode_size**
- **TypeCode_free**
- **TypeCode_print**

TypeCode_size

This function provides the size of an instance of the abstract data type described by a given **TypeCode**.

Syntax

```
long TypeCode_size   (TypeCode  tc, Environment  *env)
```

Parameters

tc (TypeCode)
The **TypeCode** whose instance size is desired.

env (Environment *)
A pointer to an **Environment** structure. The CORBA standard mandates the use of this structure as a standard way to return exception information when an error condition is detected.

Returns

rc (long)
The amount of memory needed to hold an instance of the data type described by a given **TypeCode**. No exceptions are raised by this function.

Remarks

The **TypeCode_size** function is used to obtain the size of an instance of the abstract data type described by a given **TypeCode**.

This function is a SOM-unique extension to the CORBA standard.

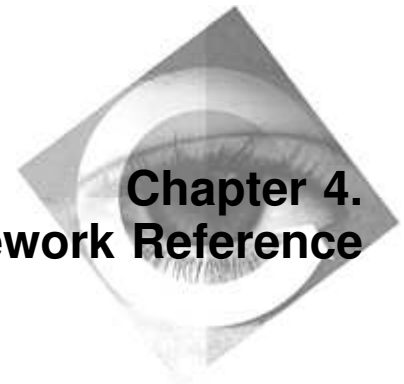
Related Information

Funtions

- **TypeCodeNew**
- **TypeCode_alignment**
- **TypeCode_equal**
- **TypeCode_kind**
- **TypeCode_param_count**
- **TypeCode_parameter**
- **TypeCode_copy**
- **TypeCode_free**

TypeCode_size

- **TypeCode_print**
- **TypeCode_setAlignment**



Chapter 4. Metaclass Framework Reference

© Copyright IBM Corp. 1994, 1996

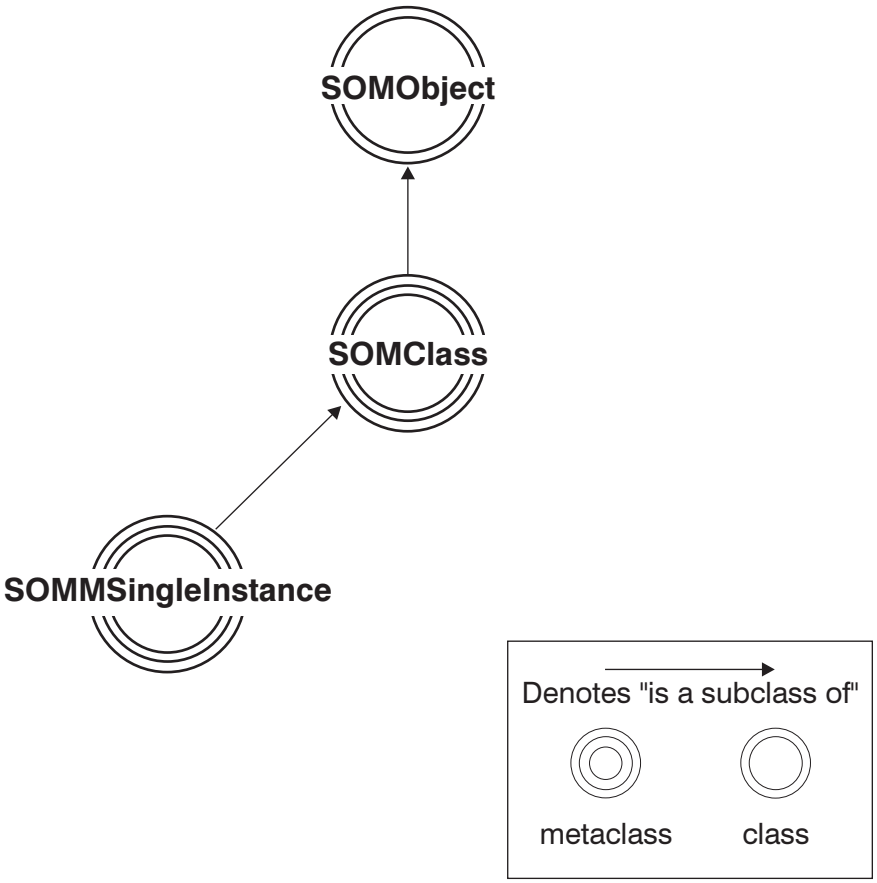


Figure 5. Metaclass class organization

SOMMBeforeAfter Metaclass

File stem: sombacls

Description

SOMMBeforeAfteris a metaclass that defines two methods **(sommBeforeMethod** and **sommAfterMethod**) which are invoke before and after each invocation of every instance method. **SOMMBeforeAfter** is designed to be subclassed. Within the subclass, each of the two methods should be overridden with a method procedure appropriate to the particular application. The before and after methods are invoked on instances (ordinary objects) of a class whose metaclass is the subclass (or child) of **SOMMBeforeAfter**, whenever any method (inherited or introduced) of the class is invoked.

Caution: The **somDefaultInit** and **somFree** methods are among th methods that get before/after behavior. This implies that the following two obligations are imposed on the programmer of a **SOMMBeforeAfter** class. First, your implementation must guard against calling the **sommBeforeMethod** before **somDefaultInit** has executed, when the object is not yet fully initialized. Second, the implementation must guard against calling **sommAfterMethod** after **somFree** at which time the object no longer exists.

New methods

None

Overridden methods

somDefaultInit, somInitMIClass

sommAfterMethod

Specifies a method that is automatically called after execution of each client Method.

Syntax

```
void sommAfterMethod  (SOMMBeforeAfter  receiver,
                       Environment  *env, SOMObject  object,
                       somId  methodId, void  *returnedvalue,
                       va_list  ap)
```

Parameters

receiver (SOMMBeforeAfter)
A pointer to an object (class) of metaclass **SOMMBeforeAfter** representing the class object that supports the method (such as, "myMethod") for which the "after" method will apply.

env (Environment *)
A pointer where the method can return exception information if an error is encountered. The dispatch method of **SOMMBeforeAfter** sets this parameter to NULL before dispatching the first **sommBeforeMethod**.

object (SOMObject)
A pointer to the instance of the receiver on which the method is invoked.

methodId (somId)
The SOM ID of the method (such as, "myMethod") that was invoked.

returnedvalue (void *)
A pointer to the value returned by invoking the method ("myMethod") on an object.

ap (va_list)
The list of input arguments to the method ("myMethod").

Returns

rc (void)

Remarks

The **sommAfterMethod** specifies a method that is automatically called after execution of each client method. The **sommAfterMethod** method is introduced in the **SOMMBeforeAfter** metaclass. The default implementation does nothing until it is overridden. The **sommAfterMethod** method is not called directly by the user. To define the desired "after" method, **sommAfterMethod** must be overridden in a metaclass that is a subclass (child) of the **SOMMBeforeAfter** metaclass.

Caution: **somFree** is among the methods that get before/after behavior, which implies that the following obligation is imposed on the programmer of a **sommAfterMethod**. Specifically, care must be taken to guard against **sommAfterMethod** being called after **somFree**, at which time the object no longer exists.

The following figure shows an invocation of "myMethod" on "myObject". Because "myObject" is an instance of a class whose metaclass is a subclass of **SOMMBeforeAfter**, "myMethod" is followed by an invocation of **sommAfterMethod** (which is shown in smaller type to denote that the user does not actually code the method). The adjacent figure illustrates the meaning of the parameters to **sommAfterMethod.**

Original Class

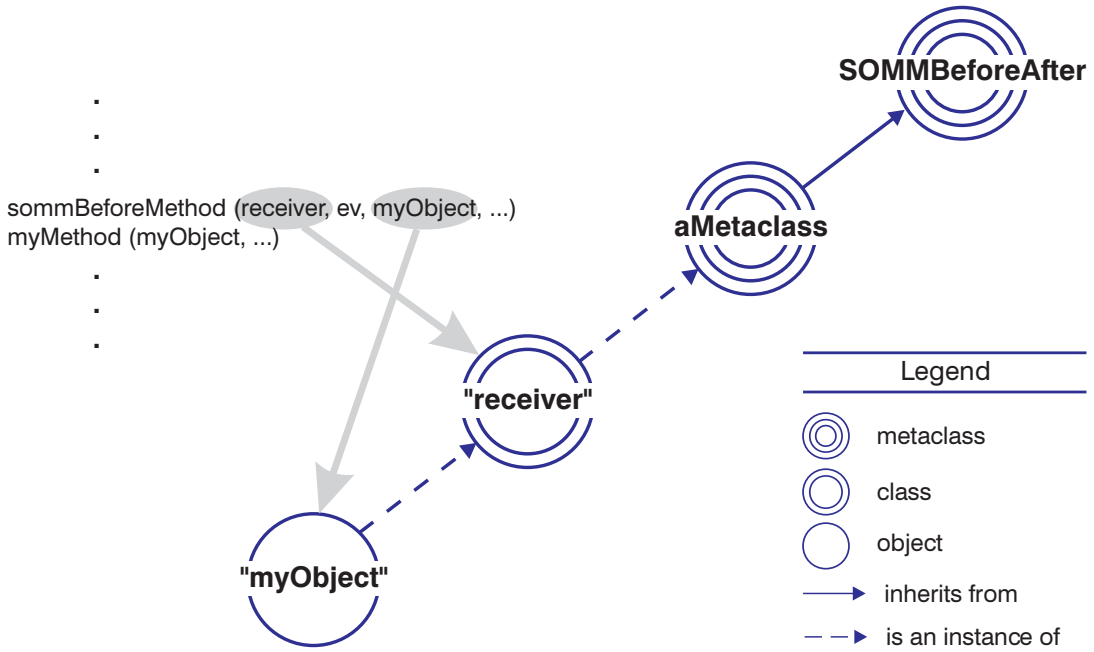
SOMMBeforeAfter

Related Methods

Methods

- **sommBeforeMethod**

sommAfterMethod



sommBeforeMethod

Specifies a method that is automatically called before execution of each client method.

Syntax

```
boolean sommBeforeMethod  (SOMMBeforeAfter  receiver,
                           Environment  *env, SOMObject  object,
                           somID  methodId, va_list  ap)
```

Parameters

receiver (SOMMBeforeAfter)
A pointer to an object (class) of metaclass **SOMMBeforeAfter** representing the class object that supports the method (such as, “myMethod”) for which the “before” method will apply.

env (Environment *)
A pointer where the method can return exception information if an error is encountered. The dispatch method of **SOMMBeforeAfter** sets this parameter to NULL before dispatching the first **sommBeforeMethod**.

object (SOMObject)
A pointer to the instance of the receiver on which the method is invoked.

methodId (somID)
The SOM Id of the method (such as, “myMethod”) that was invoked.

ap (va_list)
The list of input arguments to the method (“myMethod”).

Returns

rc (boolean)
A boolean that indicates whether or not before/after dispatching should continue. If the value is TRUE, normal before/after dispatching continues. If the value is FALSE, the dispatching skips to the sommAfterMethod associated with the preceding sommBeforeMethod. This implies that the sommBeforeMethod must do any post-processing that might otherwise be done by the sommAfterMethod. Because before/after methods are paired within a SOMMBeforeAfter metaclass,

this designe liminates the complexity of communicating to the sommAfterMethod that the sommBeforeMethod returned FALSE.

Remarks

The **sommBeforeMethod** specifies a method that is automatically called before execution of each client method. The **sommBeforeMethod** method is not called directly by the user. To define the desired "before" method, **sommBeforeMethod** must be overridden in a metaclass that is a subclass (child) of **SOMMBeforeAfter**. The default implementation does nothing until it is overridden.

Caution: **somDefaultInit** is among the methods that get before/after behavior, which implies that the following obligation is imposed on the programmer of a **sommBeforeMethod**. Specifically, care must be taken to guard against **sommBeforeMethod** being called before the **somDefaultInit** method has executed and the object is not yet fully initialized.

The following figure shows an invocation of "myMethod" on "myObject". Because "myObject" is an instance of a class whose metaclass is a subclass of **SOMMBeforeAfter**, "myMethod" is preceded by an invocation of sommBeforeMethod (which is shown in smaller type to denote that the user does not actually code the method). The adjacent figure illustrates the meaning of the parameters to **sommBeforeMethod**.

Original Class

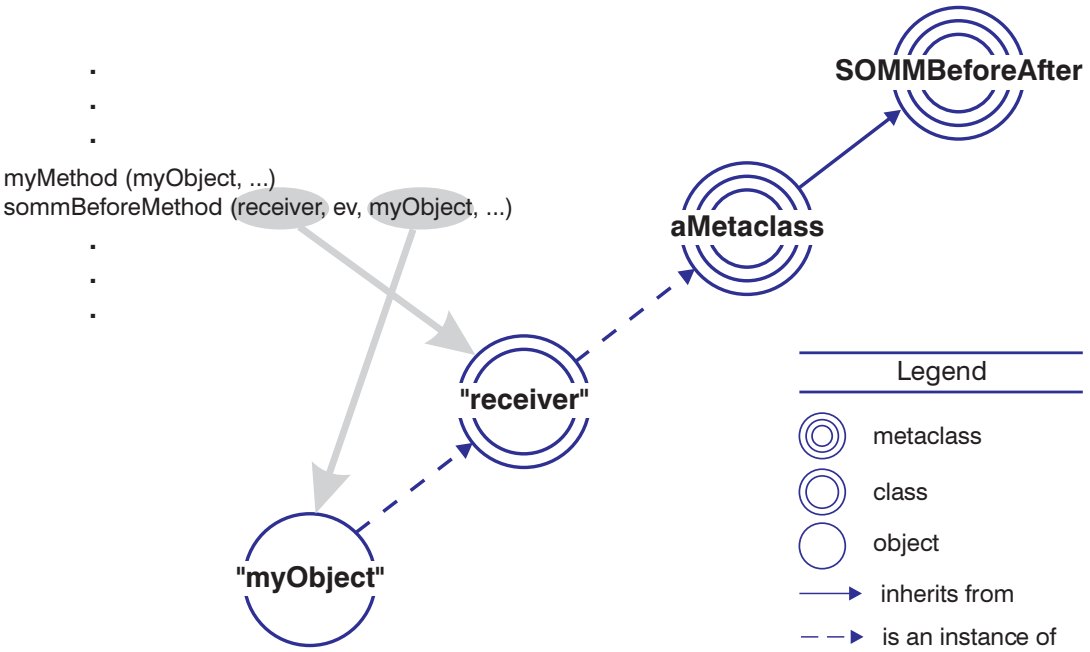
SOMMBeforeAfter

Related Methods

Methods

- **sommAfterMethod**

sommBeforeMethod



SOMMSingleInstance

File stem: snglicls

Base

SOMClass

Metaclass

SOMClass

Ancestor Classes

SOMClass
    SOMObject

Description

SOMMSingleInstance is a metaclass provided with the SOM Toolkit. It can be specified as the metaclass when defining a class for which only one instance can ever be created. The first call to <*className*>**New** in C, the **new** operator in C++, or the **somNew** method creates the one possible instance of the class. Thereafter, any subsequent "new" calls return the first (and only) instance. SOMMSingleInstance is thread safe.

Alternatively, the *method* **sommGetSingleInstance** can be used to accomplish the same purpose. The method offers an advantage in that the call site explicitly shows that something special is occurring and that a new object is not necessarily being created.

New methods

The following list shows all the SOMMSingleInstance methods.

- sommGetSingleInstance

Overridden methods

The following list shows all the methods overridden by the SOMMSingleInstance class. These methods are overridden in order to modify the behavior defined by an ancestor class.

- somInit
- somNew

sommGetSingleInstance

This method gets the one instance of a specified class for which only a single instance can exist.

Syntax

```
SOMObject sommGetSingleInstance  (SOMMSingleInstance  receiver,
                                  Environment  *env)
```

Parameters

receiver (SOMMSingleInstance)
A pointer to an object (class) whose metaclass is **SOMMSingleInstance** (or is a subclass of it).

env (Environment *)
A pointer where the method can return exception information if an error is encountered.

Returns

rc (SOMObject)
Returns a pointer to the single instance of the specified class.

Remarks

The **sommGetSingleInstance** method gets a pointer to the one instance of a class for which only a single instance can exist. A class can have only a single instance when its metaclass is the **SOMMSingleInstance** metaclass (or is a subclass of it).

The first call to *<className>***New** in C, the **new** operator in C++, or the **somNew** method creates the one possible instance of the class. Thereafter, any subsequent "new" calls return the first (and only) instance. Using the **sommGetSingleInstance** method, however, offers an advantage in that the call site explicitly shows that something special is occurring and that a new object is not necessarily being created. (That is, the **sommGetSingleInstance** method creates the single instance if it does not already exist.)

sommGetSingleInstance

Original Class

SOMMGetSingleInstance

Example Code

```
x1  = XXXNew();
x2  = XXXNew();
assert( x1 == x2 );
x3 = _sommGetSingleInstance(  _somGetClass( x1 ), env );
assert( x2 == x3 );
```

Note that the method **sommGetSingleInstance** is invoked on the class object, because **sommGetSingleInstance** is a method introduced by the metaclass **SOMMSingleInstance.**

SOMMTraced Metaclass

File stem: somtrcls

Base

Base Class

Ancestor Classes

SOMMBeforeAfter, SOMClass, SOMObject

Description

SOMMTraced is a metaclass that facilitates tracing of method invocations. Whenever a method (inherited or introduced) is invoked on an instance (simple object) of a class whose metaclass is **SOMMTraced**, a message prints to standard output giving the method parameters; then, after completion, a second message prints giving the returned value.

There is one more step for using **SOMMTraced:** nothing print unless the environment variable **SOMM_TRACED** is set. If it is set to the empty string, all traced classes print. If the environment variable SOMM_TRACED is not the empty string, it should be set to the list of names of classes that should be traced. For example, for csh users, the following command turns on printing of the trace for "Collie" and "Chihuahua", but not for any other traced class:

setenv SOMM_TRACED "Collie Chihuahua"

SOMMTraced is thread-safe.

Attributes

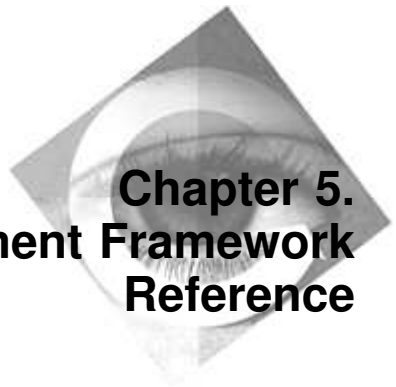
Boolean SOMMTraceIsOn. This attribute indicates whether or not tracing is turned on for a class. This gives dynamic control over the trace facility.

New methods

None

Overridden methods

somInitMIClass, sommBeforeMethod, sommAfterMethod



Chapter 5. Event Management Framework Reference

© Copyright IBM Corp. 1994, 1996

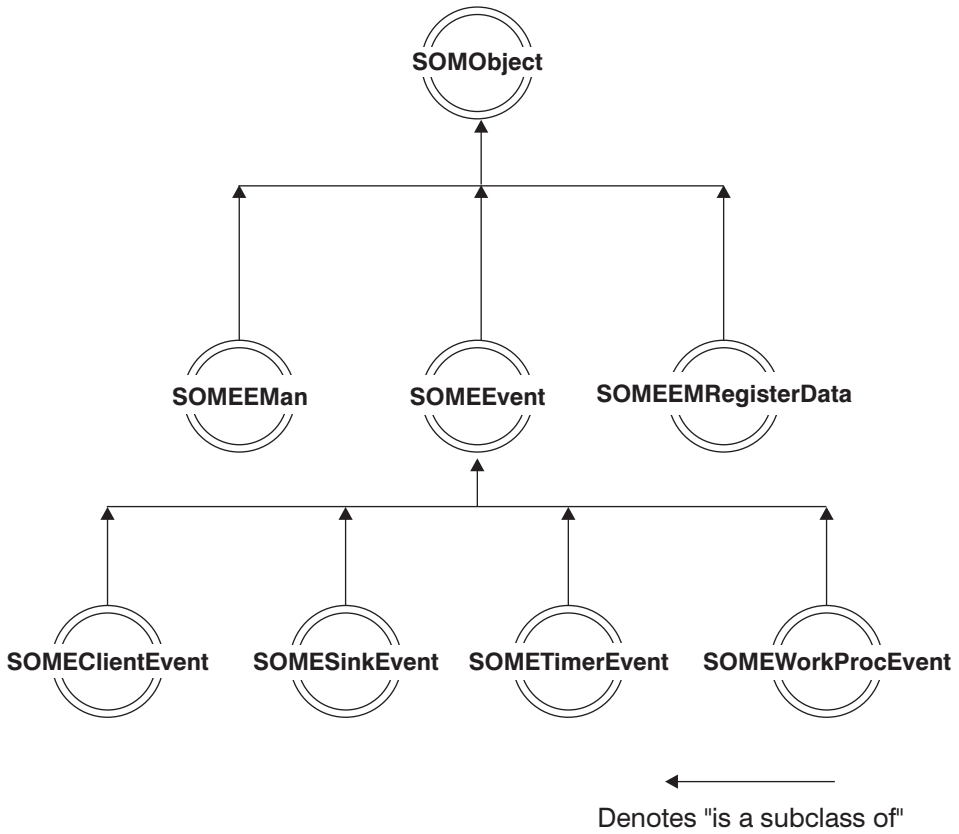


Figure 6. Event Management Framework Class Organization

SOMEClientEvent

File stem: clientev

Base

SOMEEvent

Metaclass

SOMClass

Ancestor Classes

SOMEEvent
    SOMObject

Description

This class describes generic client events within the Event Manager. Client Events are defined, created, processed and destroyed entirely by the application. The application can queue several types of client events with EMan. When a client event occurs, EMan passes an instance of this class to the callback routine. The callback can query this object about its type and obtain any event-specific information.

New methods

The following list shows all the SOMEClientEvent methods.

- somevGetEventClientData
- somevGetEventClientType
- somevSetEventClientData
- somevSetEventClientType

Overridden methods

The following list shows all the methods overridden by the SOMEClientEvent class. These methods are overridden in order to modify the behavior defined by an ancestor class.

- somInit

somevGetEventClientData

This method returns the user-defined data associated with a client event.

Syntax

```
void * somevGetEventClientData (SOMEClientEvent receiver,
        Environment *env)
```

Parameters

receiver (SOMEClientEvent)
A pointer to an object of class **SOMEClientEvent**.

env (Environment *)
A pointer to the **Environment** structure for the calling method.

Returns

rc (void *)
A pointer to user-defined client event data.

Remarks

This method returns the user-defined data (if any) associated with the Client Event object. This associated data for a given client event type is passed to EMan at the time of registration.

Original Class

SOMEClientEvent

Related Methods

Methods

- **somevSetEventClientData**

somevGetEventClientType

This method returns the type name of a client event.

Syntax

```
string somevGetEventClientType  (SOMEClientEvent  receiver,
                                 Environment  *env)
```

Parameters

receiver (SOMEClientEvent)
A pointer to an object of class **SOMEClientEvent**.

env (Environment *)
A pointer to the **Environment** structure for the calling method.

Returns

rc (string)
A null terminated string identifying the client event type.

Remarks

This method returns the client event type of the Client Event object. Client event type is a string name assigned to the event by the application at the time of registering the event.

Original Class

SOMEClientEvent

Related Methods

Methods

- **somevSetEventClientType**

somevSetEventClientData

This method sets the user-defined data of a client event.

Syntax

```
void somevSetEventClientData  (SOMEClientEvent  receiver,
                               Environment  *env, void  *clientData)
```

Parameters

receiver (SOMEClientEvent)
A pointer to an object of class **SOMEClientEvent**.

env (Environment *)
A pointer to the **Environment** structure for the calling method.

clientData (void *)
A pointer to user-defined data for this client event.

Returns

rc (void)

Remarks

This method sets the user-defined event data (if any) of the Client Event object. This associated data for a given client event type is passed to EMan at the time of registration.

Original Class

SOMEClientEvent

Related Methods

Methods

- **somevGetEventClientData**

somevSetEventClientType

This method sets the type name of a client event.

Syntax

```
void somevSetEventClientType  (SOMEClientEvent  receiver,
                               Environment  *env, string  clientType)
```

Parameters

receiver (SOMEClientEvent)
A pointer to an object of class **SOMEClientEvent.**

env (Environment *)
A pointer to the **Environment** structure for the calling method.

clientType (string)
A null terminated character string identifying the client event type. The contents of this string are entirely up to the user. However, while using class libraries that also use client events one must make sure that there are no name collisions.

Returns

rc (void)

Remarks

This method sets the client event type field of the Client Event object. Client event type is a string name assigned to the event by the application at the time of registering the event.

Original Class

SOMClientEvent

Related Methods

Methods

- **somevGetEventClientType**

SOMEEMan

File stem: eman

Base

SOMObject

Metaclass

SOMMSingleInstance

Ancestor Classes

SOMObject

Description

The Event Manager class (EMan for short) is used to handle several input events. The main purpose of this class is to provide a service that can do a blocked (or timed) wait on several event sources concurrently. Typically, in a main program, one registers an interest in an event type with EMan and specifies a callback (a procedure or a method) to be invoked when the event of interest occurs. After all the necessary registrations are complete, the main program ends with a call to **someProcessEvents** in EMan. This call is non-returning. Eman then waits on all registered event sources. The application is completely event driven at this point (that is, it does something only when an event occurs). The control returns to EMan after processing each event. Further registrations can be done from within the callback routines. Unregistrations can also be done from within the callback routines.

For applications that want to have their own main loop, EMan provides a non-blocking call (the **someProcessEvent** method), which processes just one event (if any) and returns to the main loop immediately. Note that when this call is the only one in the application's main loop, CPU cycles are wasted in constantly polling for events. In this situation, the non-returning form of the **someProcessEvents** call is preferable.

AIX Specifics:
On AIX this event manager supports Timer, Sink (any file, pipe, socket, or Message Queue), Client and WorkProc events.

OS/2 Specifics:
On OS/2 this event manager supports Timer, Sink (sockets only), Client, and WorkProc events.

Thread Safety:

To cope with multi-threaded applications on OS/2, the event-manager methods are mutually exclusive (that is, at any time only one thread can be executing inside of EMan). If an application thread needs to stop EMan from running (that is, to achieve mutual exclusion with EMan), it can use the two methods **someGetEmanSem** and **someReleaseEManSem** to acquire and release EMan semaphore(s). On AIX, since AIX does not support threads (at present), calling these two methods has no effect.

New methods

The following list shows all the SOMEEMan methods.

- someGetEmanSem
- someReleaseEManSem
- someChangeRegData
- someProcessEvent
- someProcessEvents
- someQueueEvent
- someRegister
- someRegisterEv
- someRegisterProc
- someShutdown
- someUnegister

Overridden methods

The following list shows all the methods overridden by the SOMEEMan class. These methods are overridden in order to modify the behavior defined by an ancestor class.

- somInit
- somUninit

someChangeRegData

This method changes the registration data associated with a specified registration ID.

Syntax

```
void someChangeRegData  (SOMEEMan  receiver, Environment  *env,
                         long  registrationId,
                         SOMEEMRegisterData  registerData)
```

Parameters

receiver (SOMEEMan)
A pointer to an object of class **SOMEEMan.**

env (Environment *)
A pointer to the **Environment** structure for the calling method.

registrationId (long)
The registration ID of the event interest whose data is being changed.

registerData (SOMEEMRegisterData)
A pointer to the registration data object whose contents will replace the existing registration information with EMan.

Returns

rc (void)

Remarks

This method is called to change the registration data associated with an existing registration of EMan. The existing registration is identified by the *registrationId* parameter. This ID must be the one returned by EMan when the event interest was originally registered with EMan. Further, the registration must be active (that is, it must not have been unregistered). The result of providing a non-existent or invalid registration ID is a "no op".

Original Class

SOMEEMan

Related Methods

Methods

- **someRegister**
- **SomeRegisterEv**
- **SomeRegisterProc**

Example Code

```
#include <eman.h>
SOMEEMan *EManPtr;
SOMEEMRegisterData *data;
Environment *Ev;
long RegId;

        ...
_someChangeRegData(EManPtr, Ev, RegId, data);
```

someGetEManSem

This method acquires EMan semaphore(s) to achieve mutual exclusion with EMan's activity.

Syntax

```
void someGetEManSem   (SOMEEMan  receiver, Environment  *env)
```

Parameters

receiver (SOMEEMan)
A pointer to an object of class **SOMEEMan.**

env (Environment *)
A pointer to the **Environment** structure for the calling method.

Returns

rc (void)

Remarks

When EMan is used on OS/2, multiple threads can invoke methods on EMan concurrently. EMan protects its internal data by acquiring SOM toolkit semaphore(s). The same semaphore(s) are made available to users of EMan through the methods **someGetEManSem** and **someReleaseEManSem**. If an application desires to prevent EMan event processing from interfering with its own activity (in another thread, of course), then it can call the **someGetEManSem** method and acquire EMan semaphore(s). EMan activity will resume when the application thread releases the same semaphore(s) by calling **someReleaseEManSem**.

Callers should not hold this semaphore for too long, since it essentially stops EMan activity for that duration and may cause EMan to miss some important event processing. The maximum duration for which one can hold this semaphore depends on how frequently EMan must process events.

On AIX, calling this method has no effect.

Original Class

SOMEEMan

Related Methods

Methods

- **someReleaseEManSem**

Example Code

```
#include <eman.h>
SOMEEMan *EManPtr;
Environment *Ev;

     ...
_someGetEManSem(EManPtr, Ev);
  /* Do the work that needs mutual exclusion with EMan */
_someReleaseEManSem(EManPtr, Ev);
```

someProcessEvent

This method processes one event.

Syntax

```
void someProcessEvent  (SOMEEMan receiver, Environment *env,
                        unsigned long mask)
```

Parameters

receiver (SOMEEMan)
A pointer to an object of class **SOMEEMan.**

env (Environment *)
A pointer to the **Environment** structure for the calling method.

mask (unsigned long)
A bit mask indicating the types of events to look for and process.

Returns

rc (void)

Remarks

Processes one event. This call is non-blocking. If there are no events to process it returns immediately. The mask specifies which events to process. The mask is formed by OR'ing the bit constants specified in “eventmsk.h”.

Original Class

SOMEEMan

Related Methods

Methods

- **someProcessEvents**
- **someRegister**
- **someRegisterProc**
- **someRegisterEv**

Example Code

```
#include <eman.h>

main()
{
Environment *testEnv = somGetGlobalEnvironment();
SOMEEMan *some_gEMan = SOMEEManNew();
/* Do some registrations */
   ...
while (1) {
        _someProcessEvent(some_gEMan,  testEnv,
                                EMProcessTimerEvent |
                                EMProcessSinkEvent |
                                EMProcessClientEvent );
        /***  Do other main loop work, if needed. ***/
}
} /* end of main */
```

someProcessEvents

This method processes infinite events.

Syntax

```
void someProcessEvents  (SOMEEMan  receiver, Environment  *env)
```

Parameters

receiver (SOMEEMan)
A pointer to an object of class **SOMEEMan.**

env (Environment *)
A pointer to the **Environment** structure for the calling method.

Returns

rc (void)

Remarks

This call loops forever waiting for events and dispatching them. The only way this can be broken is by calling **someShutdown** in a callback routine. It is a programming error to call this method without having registered interest in any events with EMan. Typically, a call to this method is the last statement in an application's main program.

Original Class

SOMEEMan

Related Methods

Methods

- **someProcessEvent**
- **someRegister**
- **someRegisterProc**
- **someRegisterEv**

Example Code

```
#include <eman.h>

main()
{
Environment *testEnv = somGetGlobalEnvironment();
SOMEEMan *some_gEMan = SOMEEManNew();
 /* Do some registrations */
   ...
_someProcessEvents(some_gEMan,  testEnv);
} /* end of main */
```

someQueueEvent

This method enqueues the specified client event.

Syntax

```
void someQueueEvent  (SOMEEMan  receiver, Environment  *env,
                      SOMEClientEvent  event)
```

Parameters

receiver (SOMEEMan)
A pointer to an object of class **SOMEEMan.**

env (Environment *)
A pointer to the **Environment** structure for the calling method.

event (SOMEClientEvent)

Returns

rc (void)

Remarks

Client events are defined, created, processed and destroyed by the application. EMan simply provides a means to enqueue and dequeue client events. Client events can be used in several ways. For example, if an application component wants to handle an input message arriving on a socket at a later time than when it arrives, it can receive the message in the socket callback routine, create a client event out of it, and queue it with EMan. EMan can be asked for the client event at a later time when the application is ready to handle it. Client events can also be useful to hide the origin of event sources (that is, the original event handlers receive the events and create client events in their place).

Dequeue is not a user-visible operation. Once a client event is queued, only EMan can dequeue it.

Original Class

SOMEEMan

Example Code

```
#include <eman.h>
SOMEClientEvent *clientEvent1;

clientEvent1 = SOMEClientEventNew();
/* create a client event of type "ClientType1" */
_somevSetEventClientType( clientEvent1, testEnv, "ClientType1" );
_somevSetEventClientData( clientEvent1, testEnv, "Test Msg");
  ...

/* whenever it is desired to cause this client event to happen,
   call someQueueEvent Method with this clientEvent */
_someQueueEvent(some_gEMan, env, clientEvent1);
```

someRegister

This method registers an object/method pair with EMan, given a specified *registerData* object.

Syntax

```
long someRegister  (SOMEEMan receiver, Environment *env,
                    SOMEEMRegisterData registerData,
                    SOMObject targetObject, string targetMethod,
                    void *targetData)
```

Parameters

receiver (SOMEEMan)
A pointer to an object of class **SOMEEMan.**

env (Environment *)
A pointer to the **Environment** structure for the calling method.

registerData (SOMEEMRegisterData)
A pointer to the registration data object that contains all the necessary information about the event for which an interest is being registered with EMan.

targetObject (SOMObject)
A pointer to the object that is the target of the callback method.

targetMethod (string)
The name of the callback method.

targetData (void *)
A pointer to a data structure to be passed to the callback method when the event occurs.

Returns

rc (long)
The registration ID.

Remarks

This method allows for registering an event of interest with EMan, with an object method as the callback. It is assumed that the target method has been declared as using OIDL callstyle. The event of interest and its details are filled in a registration data object *registerData*. The information about the callback routine is indicated by *targetObject* and *targetMethod*.

A mismatch between the target method's callstyle and the registration method used (that is, **someRegister** vs. **someRegisterEv**) can result in unpredictable results. Also see the **callstyle** modifier of the SOM Interface Definition Language described in Chapter 4 "SOM IDL and the SOM Compiler" of the *SOM Programming Guide*

Note: The target method is called using name-lookup method resolution.

Original Class

SOMEEMan

Related Methods

Methods

- **someRegisterEv**
- **someRegisterProc**
- **someUnRegister**

Example Code

```
#include <eman.h>
#include <emobj.h>

Environment *testEnv = somGetGlobalEnvironment();
some_gEMan = SOMEEManNew();         /* create an EMan object */
data = SOMEEMRegisterDataNew( ); /* create a reg data object */
target = EMObjectNew();    /* create a target object */

/* reRegister a timer event */
_someClearRegData( data, env );
_someSetRegDataEventMask( data, env, EMTimerEvent, NULL );
_someSetRegDataTimerInterval( data, env, 100 );
regId1 = _someRegister( some_gEMan, env, data, target,
                        "eventMethod", "Timer 100" );
```

someRegisterEv

This method registers the (object, method, **Environment** parameter) combination of a callback with EMan, given a specified *registerData* object.

Syntax

```
long someRegisterEv  (SOMEEMan receiver, Environment *env,
                      SOMEEMRegisterData registerData,
                      SOMObject targetObject,
                      Environment callbackEv, string targetMethod,
                      void *targetData)
```

Parameters

receiver (SOMEEMan)
A pointer to an object of class **SOMEEMan**.

env (Environment *)
A pointer to the **Environment** structure for the calling method.

registerData (SOMEEMRegisterData)
A pointer to registration data object that contains all the necessary information about the event for which an interest is being registered with EMan.

targetObject (SOMObject)
A pointer to the object which is the target of the callback method

callbackEv (Environment)
A pointer to the Environment structure to be passed to the callback method

targetMethod (string)
The name of the callback method.

targetData (void *)
A pointer to a data structure to be passed to the callback method when the event occurs.

Returns

rc (long)
The registration ID.

Remarks

This method allows for registering an event interest with EMan with an object method as callback. The *callbackEv* is used as the environment pointer when EMan makes the callback. It is assumed that the target method has been declared as using IDL callstyle. The event of interest and its details are filled in a registration data object *registerData*. The information about the callback routine is indicated by *targetObject* and *targetMethod*.

A mismatch in the target method's callstyle and the registration method called (**someRegister** vs. **someRegisterEv**) can result in unpredictable results. Also see the **callstyle** modifier of the SOM Interface Definition Language described in Chapter 4 "SOM IDL and the SOM Compiler" of the *SOM Programming Guide*.

Note: The target method is called using name-lookup method resolution.

Original Class

SOMEEMan

Related Methods

Methods

- **someRegister**
- **someRegisterProc**
- **someUnRegister**

Example Code

```
#include <eman.h>
#include <emobj.h>

Environment *testEnv = somGetGlobalEnvironment();
Environment *targetEv = somGetGlobalEnvironment();
some_gEMan = SOMEEManNew();         /* create an EMan object */
data = SOMEEMRegisterDataNew( ); /* create a reg data object */
target = EMObjectNew();    /* create a target object */

/* reRegister a timer event */
_someClearRegData( data, env );
_someSetRegDataEventMask( data, env, EMTimerEvent, NULL );
_someSetRegDataTimerInterval( data, env, 100 );
regId1 = _someRegisterEv( some_gEMan,env, data, target,targetEv,
                          "eventMethod", "Timer 100" );
 /* eventMethod of target is assumed to use callstyle=idl */
```

someRegisterProc

This method registers the procedure with EMan given the specified *registerData*.

Syntax

```
long someRegisterProc  (SOMEEMan  receiver, Environment  *env,
                        SOMEEMRegisterData  registerData,
                        EMRegProc  *targetProcedure, void  *targetData)
```

Parameters

receiver (SOMEEMan)
A pointer to an object of class **SOMEEMan.**

env (Environment *)
A pointer to the **Environment** structure for the calling method.

registerData (SOMEEMRegisterData)
A pointer to registration data object that contains all the necessary information about the event for which an interest is being registered with EMan.

targetProcedure (EMRegProc *)
A pointer to the procedure (callback) that is called when the registered event occurs.

targetData (void *)

Returns

rc (long)
The registration ID.

Original Class

SOMEEMan

Related Methods

Methods

- **someRegister**
- **someRegisterEv**

- **someUnRegister**

Example Code

```
#include <eman.h>

void MyCallBack(SOMEEvent *event, void *somedata){
 ...
}

Environment *testEnv = somGetGlobalEnvironment();
some_gEMan = SOMEEManNew();         /* create an EMan object */
data = SOMEEMRegisterDataNew( ); /* create a reg data object */

/* reRegister a timer event */
_someClearRegData( data, env );
_someSetRegDataEventMask( data, env, EMTimerEvent, NULL );
_someSetRegDataTimerInterval( data, env, 100 );
regId1 = _someRegisterProc( some_gEMan, env, data,
                        MyCallBack, "Timer 100" );
```

someReleaseEManSem

This method releases the semaphore obtained by the **someGetEManSem** method.

Syntax

```
void someReleaseEManSem   (SOMEEMan  receiver, Environment  *env)
```

Parameters

receiver (SOMEEMan)
A pointer to an object of class **SOMEEMan.**

env (Environment *)
A pointer to the **Environment** structure for the calling method.

Returns

rc (void)

Remarks

When EMan is used on OS/2, multiple threads can invoke methods on EMan concurrently. EMan protects its internal data by acquiring SOM toolkit semaphore(s). The same semaphore(s) are made available to users of EMan through the methods **someGetEManSem** and **someReleaseEManSem**. If an application desires to prevent EMan's event processing from interfering with its own activity (in another thread, of course), then it can call the **someGetEManSem** method and acquire EMan semaphore(s). EMan activity will resume when the application thread releases the same semaphore(s) by calling **someReleaseEManSem**.

Callers should not hold this semaphore for too long, since it essentially stops EMan activity for that duration and may cause EMan to miss some important event processing. The maximum duration for which one can hold this semaphore depends on how frequently EMan must process events.

On AIX, calling this method has no effect.

Original Class

SOMEEMan

Related Methods

Methods

- **someGetEManSem**

Example Code

```
#include <eman.h>
SOMEEMan *EManPtr;
Environment *Ev;

    ...
_someGetEManSem(EManPtr, Ev);
  /* Do the work that needs mutual exclusion with EMan */
_someReleaseEManSem(EManPtr, Ev);
```

someShutdown

This method shuts down an EMan event loop. (That is, this makes the **someProcessEvents** return!)

Syntax

```
void someShutdown   (SOMEEMan  receiver, Environment  *env)
```

Parameters

receiver (SOMEEMan)
A pointer to an object of class **SOMEEMan.**

env (Environment *)
A pointer to the **Environment** structure for the calling method.

Returns

rc (void)

Remarks

This can be called from a callback routine to break the someProcessEvents loop.

Original Class

SOMEEMan

Related Methods

Methods

- **someProcessEvents**

Example Code

```
#include <eman.h>
SOMEEMan *some_gEMan;

void MyCallBack(SOMEEvent *event, void *somedata){
 ...
_someShutdown(some_gEMan, env);
}
main()
{
Environment *testEnv = somGetGlobalEnvironment();
SOMEEMan *some_gEMan = SOMEEManNew();
 /* Do some registrations. At least one involving MyCallBack */
   ...
_someProcessEvents(some_gEMan,  testEnv);
}
```

someUnRegister

This method unregisters the event interest associated with a specified *registrationId* within EMan.

Syntax

```
void someUnRegister  (SOMEEMan  receiver, Environment  *env,
                      long  registrationId)
```

Parameters

receiver (SOMEEMan)
A pointer to an object of class **SOMEEMan.**

env (Environment *)
A pointer to the **Environment** structure for the calling method.

registrationId (long)
The registration ID of the event that needs to be unregistered.

Returns

rc (void)

Remarks

When an application is no longer interested in a given event, it can unregister the event interest from EMan. EMan will stop making callbacks on this event, even if the event source continues to be active and generates events.

Original Class

SOMEEMan

Related Methods

Methods

- **someRegister**
- **someRegisterEv**
- **someRegisterProc**

Example Code

```
#include <eman.h>
long regId1;

 ...
/* Register a timer */
regId1 = _someRegisterEv( some_gEMan,env, data, target,targetEv,
                          "eventMethod", "Timer 100" );

 ...
/* Unregister the timer */
_someUnRegister(some_gEMan, env, regId1);
```

SOMEEMRegisterData

File stem: emregdat

Base

SOMObject

Metaclass

SOMClass

Ancestor Classes

SOMClass
    WPObject

Description

This class is used for holding registration information for event types to be registered with EMan. EMan extracts all needed information from this object and saves the information in its internal data structures. An instance of this class must be created, properly initialized, and passed to the registration methods of EMan for registering interest in any kind of event.

New methods

The following list shows all the SOMEEMRegisterData methods.

- someClearRegData
- someSetRegDataClientType
- someSetRegDataEventMask
- someSetRegDataSink
- someSetRegDataSinkMask
- someSetRegDataTimerCount
- someSetRegDataTimerInterval

Overriding methods

The following list shows all the methods overridden by the SOMEEMRegisterData class. These methods are overridden in order to modify the behavior defined by an ancestor class.

- somInit
- somUnInit

someClearRegData

This method clears the registration data.

This method initializes all fields of a RegData object to their default values.

Syntax

```
void someClearRegData  (SOMEEMRegisterData  receiver,
                        Environment  *env)
```

Parameters

receiver (SOMEEMRegisterData)
A pointer to an object of class **SOMEEMRegisterData**.

env (Environment *)
A pointer to the **Environment** structure for the calling method.

Returns

rc (void)

Original Class

SOMEEMRegisterData

someSetRegDataClientType

This method sets the type name for a client event.

Syntax

```
void someSetRegDataClientType  (SOMEEMRegisterData  receiver,
                                Environment  *env, string  clientType)
```

Parameters

receiver (SOMEEMRegisterData)
A pointer to an object of class **SOMEEMRegisterData**.

env (Environment *)
A pointer to the **Environment** structure for the calling method.

clientType (string)
A null-terminated character string identifying the client event type. The contents of this string are entirely up to the user. However, while using class libraries that also use client events, one must make sure that there are no name collisions.

Returns

rc (void)

Remarks

Client events are defined, created, processed, and destroyed entirely by the application. The application can queue several types of client events with EMan. This method sets the client event type field of the registration data object. Thus, this information is communicated to EMan, helping it deal with enqueueing and dequeing the different client events.

Original Class

SOMEEMRegisterData

Related Methods

Methods:

- **someClearRegData**

someSetRegDataEventMask

This method sets the generic event mask within the registration data using NULL terminated event type list.

Syntax

```
void someSetRegDataEventMask  (SOMEEMRegisterData  receiver,
                               Environment  *env, long  eventType,
                               va_list  ap)
```

Parameters

receiver (SOMEEMRegisterData)
A pointer to an object of class **SOMEEMRegisterData**.

env (Environment *)
A pointer to the **Environment** structure for the calling method.

eventType (long)
A bit constant indicating the type of event being registered with EMan.

ap (va_list)
Additional event types (usually NULL).

Returns

rc (void)

Remarks

This allows setting the event mask within the registration data object. Essentially, this tells EMan what kind of event is being registered with it. The event type list is a series of constants defined in eventmsk.h. Although the current interface supports a NULL terminated list of event types, currently each registration with EMan names only one event type. Thus, one usually gives only one named constant as the event type and follows it with a NULL parameter.

someSetRegDataEventMask

Original Class

SOMEEMRegisterData

Related Methods

Methods:

- **someSetRegDataSink**
- **someClearRegData**

Example Code

```
#include <eman.h>
long regId1;
int msgsock;

 ...
/* Register msgsock socket with EMan for further communication */
_someClearRegData( data, env );
_someSetRegDataEventMask( data, env, EMSinkEvent, NULL );
/* The above call enables EMan to know (during registration) that
we are talking about a Sink Event */
_someSetRegDataSink( data, env, msgsock );
_someSetRegDataSinkMask( data, env, EMInputReadMask);

regId = _someRegisterProc( some_gEMan, env, data,
                        ReadSocketAndPrint, "READMSG" );
```

someSetRegDataSink

This method sets the file descriptor (or socket ID, or message queue ID) for the sink event.

Syntax

```
void someSetRegDataSink  (SOMEEMRegisterData  receiver,
                          Environment  *env, long  sink)
```

Parameters

receiver (SOMEEMRegisterData)
A pointer to an object of class **SOMEEMRegisterData**.

env (Environment *)
A pointer to the **Environment** structure for the calling method.

sink (long)
An integer value indicating the file descriptor for input/output. It can also be a socket ID, pipe ID or a message queue ID.

Returns

rc (void)

Remarks

This method enables setting the true type of an event object. Typically, a subclass of Event calls this method (or overrides this method) to set the event type to indicate its true class(type).

Original Class

SOMEEMRegisterData

Related Methods

Methods:

- **someClearRegData**

someSetRegDataSinkMask

This method sets the sink mask within the registration data object.

Syntax

```
void someSetRegDataSinkMask  (SOMEEMRegisterData  receiver,
                              Environment  *env,
                              unsigned  long  sinkmask)
```

Parameters

receiver (SOMEEMRegisterData)
A pointer to an object of class **SOMEEMRegisterData**.

env (Environment *)
A pointer to the **Environment** structure for the calling method.

sinkmask (unsigned long)
A bit mask indicating the types of events of interest on a given sink.

Returns

rc (void)

Remarks

The sink mask within the registration data allows one to express interest in different events of the same event source. For example, using this mask one can express interest in being notified when there is input for reading, when the resource is ready for writing output, or just when exceptions occur.

Original Class

SOMEEMRegisterData

Related Methods

Methods:

- **someSetRegDataSink**
- **someClearRegData**

Example Code

```
#include <eman.h>
long regId1;
int msgsock;

 ...
/* Register msgsock socket with EMan for further communication */
_someClearRegData( data, env );
_someSetRegDataEventMask( data, env, EMSinkEvent, NULL );
_someSetRegDataSink( data, env, msgsock );
_someSetRegDataSinkMask( data, env,
                         EMInputReadMask|EMInputExceptMask);
/* The above call expresses interest in knowing when there is
 input to be read from the socket and when there is an exception
condition associated with this socket. */
regId = _someRegisterProc( some_gEMan, env, data,
                        ReadSocketAndPrint, "READMSG" );
```

someSetRegDataTimerCount

This method sets the number of times the timer will trigger, within the registration data.

Syntax

```
void someSetRegDataTimerCount  (SOMEEMRegisterData  receiver,
                                Environment  *env, long  count)
```

Parameters

receiver (SOMEEMRegisterData)
A pointer to an object of class **SOMEEMRegisterData**.

env (Environment *)
A pointer to the **Environment** structure for the calling method.

count (long)
An integer indicating the number of times the timer event has to occur.

Returns

rc (void)

Remarks

The **someSetRegDataTimerCount** method sets the number of times the timer will trigger, within the registration data. The default behavior is for the timer to trigger indefinitely.

Original Class

SOMEEMRegisterData

Related Methods

Methods:

- **someClearRegData**

Example Code

```
#include <eman.h>
long regId1;

 ...
/* Register a timer */
_someClearRegData( data, env );
_someSetRegDataEventMask( data, env, EMTimerEvent, NULL );
_someSetRegDataTimerInterval( data, env, 100 );
_someSetRegDataTimerCount(data, env, 1);
/* make this a one time timer event */
regId1 =  _someRegister( some_gEMan,env, data, target,
                         "eventMethod", "Timer 100" );
```

someSetRegDataTimerInterval

This method sets the timer interval within the registration data.

Syntax

```
void someSetRegDataTimerInterval  (SOMEEMRegisterData  receiver,
                                   Environment  *env, long  interval)
```

Parameters

receiver (SOMEEMRegisterData)
A pointer to an object of class **SOMEEMRegisterData**.

env (Environment *)
A pointer to the **Environment** structure for the calling method.

interval (long)
An integer indicating the timer interval in milliseconds.

Returns

rc (void)

Remarks

This call allows setting the timer interval (in milliseconds) within the registration data object.

Original Class

SOMEEMRegisterData

Related Methods

Methods:

- **someClearRegData**

Example Code

```
#include <eman.h>
long regId1;

 ...
/* Register a timer */
_someClearRegData( data, env );
_someSetRegDataEventMask( data, env, EMTimerEvent, NULL );
_someSetRegDataTimerInterval( data, env, 100 );
/* Sets the timer interval to 100 milliseconds */
regId1 = _someRegister( some_gEMan,env, data, target,
                        "eventMethod", "Timer 100" );
```

SOMEEvent

File stem: event

Base

SOMObject

Metaclass

SOMClass

Ancestor Classes

SOMClass

SOMObject

Description

This is the base class for all generic events within the Event Manager. It simply temestamps an event before it is passed to a callback routine. The event type is set to the true type by a subclass. The types currently used by the Event Management Framework are defined in eventmsk.h. Any subclass of this class must avoid name and value collisions with eventmsk.h.

New methods

The following list shows new methods defined for the SOMEEvent class.

- somevGetEventTime
- somevGetEventType
- somevSetEventTime
- somevSetEventType

Overridden methods

The following list shows all the methods overridden by the SOMEEvent class. These methods are overridden in order to modify the behavior defined by an ancestor class.

- somInit

somevGetEventTime

Returns the time of the generic event in milliseconds.

Syntax

```
unsigned long somevGetEventTime  (SOMEEvent  receiver,
                                  Environment  *env)
```

Parameters

receiver (SOMEEvent)
A pointer to an object of class **SOMEEvent.**

env (Environment *)
A pointer to the **Environment** structure for the calling method.

Returns

rc (unsigned long)
An event timestamp in milliseconds.

Remarks

Eman timestamps every event before dispatching it. The current time is obtained from the operating system (for example, using a 'gettimeofday' call), is converted to milliseconds, and is given as the value of the timestamp. When this function is called, the event timestamp is returned.

Original Class

SOMEEvent

Related Methods

Methods

- somevSetEventTime

somevGetEventType

Returns the type of the generic event.

Syntax

```
unsigned long somevGetEventType  (SOMEEvent  receiver,
                                  Environment  *env)
```

Parameters

receiver (SOMEEvent)
A pointer to an object of class **SOMEEvent.**

env (Environment *)
A pointer to the **Environment** structure for the calling method.

Returns

rc (unsigned long)
A type value (an integer constant defined in eventmsk.h).

Remarks

This method returns the true type of a given event object (for example, to identify the particular subclass of the event object). The type is an integer valued constant defined in eventmsk.h.

Original Class

SOMEEvent

Related Methods

Methods

- somevSetEventType

somevSetEventTime

Sets the time of the generic event (time is in milliseconds).

Syntax

```
void somevSetEventTime  (SOMEEvent  receiver, Environment  *env,
                         unsigned long  time)
```

Parameters

receiver (SOMEEvent)
A pointer to an object of class **SOMEEvent.**

env (Environment *)
A pointer to the **Environment** structure for the calling method.

time (unsigned long)

Returns

rc (void)

Remarks

EMan timestamps every event before dispatching it. The current time is obtained from the operating system (for example, using a 'gettimeofday' call), converted to milliseconds, and is given as the value of the timestamp. When an event occurs, EMan sets the timestamp of the event by calling this method.

Original Class

SOMEEvent

Related Methods

Methods

- somevGetEventTime

somevSetEventType

Sets the type of the generic event.

Syntax

```
void somevSetEventType  (SOMEEvent  receiver, Environment  *env,
                         unsigned long  type)
```

Parameters

receiver (SOMEEvent)
A pointer to an object of class **SOMEEvent.**

env (Environment *)
A pointer to the **Environment** structure for the calling method.

type (unsigned long)
An integer value indicating the type of the event (a constant defined in eventmsk.h).

Returns

rc (void)

Remarks

This method enables setting the true type of an event object. Typically, a subclass of **SOMEEvent** calls this method (or overrides this method) to set the event type to indicate its true type.

Original Class

SOMEEvent

Related Methods

Methods

- somevGetEventType

SOMESinkEvent

File stem: sinkev

Base

SOMEEvent

Metaclass

SOMClass

Ancestor Classes

SOMEEvent
    SOMObject

Description

This class describes a sink event that is generated by EMan when it notices activity on a registered sink. On AIX, a sink refers to any file descriptor (file open for reading or writing), any pipe descriptor, a socket ID or a message queue ID. On OS/2, a sink refers to a socket ID. One can register for three types of interest in a sink: Read interest, Write interest, and Exception interest. (See eventmsk.h file to determine the appropriate bit constants and see method **someSetRegDataSinkMask** for their use.)

EMan passes an instance of this class as a parameter to the callback registered for Sink Events. The callback can query the instance for some information on the sink.

New methods

The following list shows all the SOMESinkEvent methods.

- somevGetEventSink
- somevSetEventSink

Overridden methods

The following list shows all the methods overridden by the SOMESinkEvent class. These methods are overridden in order to modify the behavior defined by an ancestor class.

- somInit

somevGetEventSink

This method returns the sink, or source of I/O, of the generic sink event.

Syntax

```
long somevGetEventSink   (SOMESinkEvent  receiver, Environment  *env)
```

Parameters

receiver (SOMESinkEvent)
A pointer to an object of class **SOMESinkEvent**.

env (Environment *)
A pointer to the **Environment** structure for the calling method.

Returns

rc (long)
An integer value indicating the file descriptor for input/output. It can also be a socket ID, pipe ID or a message queue ID.

Remarks

The sink ID in the SinkEvent is returned. For message queues it is the queue ID, for files it is the file descriptor, for sockets it is the socket ID, and for pipes it is the pipe descriptor.

Original Class

SOMESinkEvent

Related Methods

Methods:

- **somevSetEventSink**

somevSetEventSink

This method sets the sink, or source of I/O, of the generic sink event.

Syntax

```
void somevSetEventSink  (SOMESinkEvent  receiver, Environment  *env,
                         long  sink)
```

Parameters

receiver (SOMESinkEvent)
A pointer to an object of class **SOMESinkEvent**.

env (Environment *)
A pointer to the **Environment** structure for the calling method.

sink (long)
An integer value indicating the file descriptor for input/output. It can also be a socket ID, pipe ID, or a message queue ID.

Returns

rc (void)

Remarks

The sink ID in the SinkEvent is set. For message queues, it is the queue ID; for files it is the file descriptor; for sockets it is the socket ID; and for pipes it is the pipe descriptor.

Original Class

SOMESinkEvent

Related Methods

Methods:

- **somevGetEventSink**

SOMETimerEvent

File stem: timerev

Base

SOMEEvent

Metaclass

SOMClass

Ancestor Classes

SOMEEvent
    SOMObject

Description

This class describes a timer event that is generated by EMan when any of its registered timers pops.

EMan passes an instance of this class as a parameter to the callbacks registered for Timer Events. The callback can query the instance for information on the timer interval and on any generic event properties.

New methods

The following list shows all the SOMETimerEvent methods.

- somevGetEventInterval
- somevSetEventInterval

Overridden methods

The following list shows all the methods overridden by the SOMETimerEvent class. These methods are overridden in order to modify the behavior defined by an ancestor class.

- somInit

somevGetEventInterval

This method returns the interval of the generic timer event (time in milliseconds).

Syntax

```
void somevGetEventInterval  (SOMETimerEvent  receiver,
                             Environment  *env)
```

Parameters

receiver (SOMETimerEvent)
A pointer to an object of class **SOMETimerEvent.**

env (Environment *)
A pointer to the **Environment** structure for the calling method.

Returns

rc (void)
The interval time in milliseconds.

Remarks

The **somevGetEventInterval** method returns the interval of the generic timer event (time in milliseconds).

Original Class

SOMETimerEvent

Related Methods

Methods

- **somevSetEventInterval**

somevSetEventInterval

This method sets the interval of the generic timer event (in milliseconds).

Syntax

```
void somevSetEventInterval  (SOMETimerEvent  receiver,
                             Environment  *env, long  interval)
```

Parameters

receiver (SOMETimerEvent)
A pointer to an object of class **SOMETimerEvent.**

env (Environment *)
A pointer to the **Environment** structure for the calling method.

interval (long)
The timer interval in milliseconds.

Returns

rc (void)

Remarks

The **somevSetEventInterval** method sets the interval of the generic timer event (in milliseconds).

Original Class

SOMETimerEvent

Related Methods

Methods

- **somevGetEventInterval**

SOMEWorkProcEvent

File stem: workprev

Base

SOMEEvent

Metaclass

SOMClass

Ancestor Classes

SOMEEvent
    SOMObject

Description

This class describes a work procedure event object. It currently has no methods of its own. However, it sets the event type in its super class to say "EMWorkProcEvent" to help identify itself. These events are created and dispatched by EMan when a work procedure (something that the application wants to run when no other events are happening) is registered with EMan.

EMan passes an instance of this class as a parameter to the callback registered for WorkProc Events.

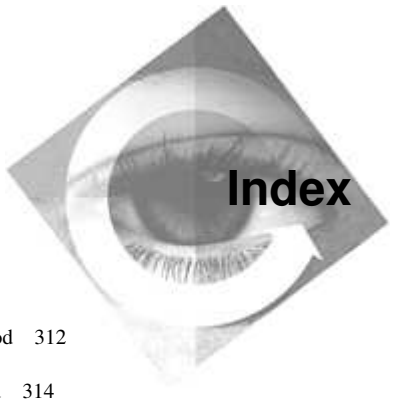
New methods

There are currently no new methods defined for the SOMEWorkProcEvent class.

Overridden methods

The following list shows all the methods overridden by the SOMEWorkProcEvent class. These methods are overridden in order to modify the behavior defined by an ancestor class.

- somInit



Index

A

activate_impl_failed 503
activate_impl_failed method 503
add_arg 406
add_arg method 406
add_class_to_impldef 344
add_class_to_impldef method 344
add_impldef 346
add_impldef method 346
add_item 367
add_item method 367

B

BOA class 305

C

change_id 505
change_id method 505
change_implementation 307
change_implementation method 307
contents 531
contents method 531
Context class 328
Context_delete Macro 301
create 309
create method 309
create_child 329
create_child method 329
create_constant 507
create_constant method 507
create_list 394
create_list method 394
create_operation_list 396
create_operation_list method 396
create_request 437
create_request method 437
create_request_args 441
create_request_args method 441
create_SOM_ref 509
create_SOM_ref method 509

D

deactivate_impl 312
deactivate_impl method 312
deactivate_obj 314
deactivate_obj method 314
delete_impldef 348
delete_impldef method 348
delete_values 331
delete_values method 331
describe 525
describe method 525
describe_contents 534
describe_contents method 534
describe_interface 544
describe_interface method 544
destroy 333, 409
destroy method (Context object) 333
destroy method (Request object) 409
dispose 316
dispose method 316
DSOM Framework 283
- BOA class 305, 307, 309, 312, 314, 316, 318, 320, 322, 324, 326
- Context class 328, 329, 331, 333, 335, 337, 339
- Functions 285, 287, 289, 293, 297, 299
  - get_next_response function 285
  - ORBfree function 287
  - send_multiple_requests function 289
  - SOMD_Init function 293
  - SOMD_RegisterCallback function 297
  - SOMD_Uninit function 299
- ImplementationDef class 341
- ImplRepository class 343, 344, 346, 348, 352, 354, 356, 358, 362, 364
  - add_class_to_impldef method 344
  - add_impldef method 346
  - delete_impldef method 348
  - find_classes_by_impldef method 352
  - find_impldef method 354
  - find_impldef_by_alias method 356
  - find_impldef_by_class method 358
  - remove_class_from_impldef method 362
  - update_impldef method 364
- NVList class 367, 369, 371, 373, 375, 378
- ObjectMgr class 383, 385, 387, 389, 391

© Copyright IBM Corp. 1994, 1996

DSOM Framework *(continued)*
ORB class 394, 396, 398, 400, 402
Request class 406, 409, 411, 413, 417
SOMDClientProxy class 421, 423, 425, 429, 431, 433
SOMDObject class 437, 441, 445, 447, 449, 451, 453, 455, 457, 459
SOMDObjectMgr class 461, 463, 465, 467, 469
somdFindAnyServerByClass method 463
somdFindServer method 465
somdFindServerByName method 467
somdFindServersByClass method 469
SOMDServer class 472, 474, 476, 478, 482, 484
somdCreateObj method 472
somdDeleteObj method 474
somdDispatchMethod method 476
somdGetClassObj method 478
somdRefFromSOMObj method 482
somdSOMObjFromRef method 484
SOMOA class 501, 503, 505, 507, 509, 511, 513, 515
activate_impl_failed method 503
change_id method 505
create_constant method 507
create_SOM_ref method 509
execute_next_request method 511
execute_request_loop method 513
get_SOM_object method 515
duplicate 445
duplicate method 445

E

EMan 599
Event Management Framework 599
SOMEClientEvent class 601, 602, 603, 604, 605
somevGetEventClientData method 602
somevGetEventClientType method 603
somevSetEventClientData method 604
somevSetEventClientType method 605
SOMEEMan class 606, 608, 610, 612, 614, 616, 618, 620, 622, 624, 626, 628
someChangeRegData method 608
someGetEManSem method 610
someProcessEvent method 612
someProcessEvents method 614
someQueueEvent method 616
someRegister method 618
someRegisterEv method 620
someRegisterProc method 622
someReleaseEManSem method 624
someShutdown method 626
someUnRegister method 628
Event Management Framework *(continued)*
SOMEEMRegisterData class 630, 631, 632, 633, 635, 636, 638, 640
someClearRegData method 631
someSetRegDataClientType method 632
someSetRegDataEventMask method 633
someSetRegDataSink method 635
someSetRegDataSinkMask method 636
someSetRegDataTimerCount method 638
someSetRegDataTimerInterval method 640
SOMEEvent class 642, 643, 644, 645, 646
SOMESinkEvent class 647, 648, 649
somevGetEventSink method 648
somevSetEventSink method 649
SOMETimerEvent class 650, 651, 652
somevGetEventInterval method 651
somevSetEventInterval method 652
SOMEWorkProcEvent class 653
execute_next_request 511
execute_next_request method 511
execute_request_loop 513
execute_request_loop method 513

F

find_all_impldefs 350
find_classes_by_impldef 352
find_classes_by_impldef method 352
find_impldef 354
find_impldef method 354
find_impldef_by_alias 356
find_impldef_by_alias method 356
find_impldef_by_class 358
find_impldef_by_class method 358
free 369
free method 369
free_memory 371
free_memory method 371

G

get_count 373
get_count method 373
get_default_context 398
get_default_context method 398
get_id 318
get_id method 318
get_implementation 447
get_implementation method 447

get_interface 449
get_interface method 449
get_item 375
get_item method 375
get_next_response 285
get_next_response function 285
get_principal 320
get_principal method 320
get_response 411
get_response method 411
get_SOM_object 515
get_SOM_object method 515
get_values 335
get_values method 335

I

impl_is_ready 322
impl_is_ready method 322
ImplementationDef class 341
ImplRepository class 343
Interface Repository Framework 517
  Contained class 525, 528
  Container class 531, 534, 537
  Functions 572
  InterfaceDef class 544
  Repository class 552, 554, 556
  TypeCode... functions 560, 561, 563, 565, 567, 572, 576, 577, 580, 582, 583
invoke 413
invoke method 413
is_constant 451
is_constant method 451
is_nil 453
is_nil method 453
is_proxy 455
is_proxy method 455
is_SOM_ref 457
is_SOM_ref method 457

L

lookup_id 552
lookup_id method 552
lookup_modifier 554
lookup_modifier method 554
lookup_name 537
lookup_name method 537

M

Metaclass classes/methods 585
  SOMMSingleInstance class 594, 595

O

obj_is_ready 324
obj_is_ready method 324
object_to_string 400
object_to_string method 400
ORBfree 287
ORBfree function 287

R

release 459
release method 459
release_cache 556
release_cache method 556
remove_class_from_all 360
remove_class_from_impldef 362
remove_class_from_impldef method 362
Request_delete Macro 303

S

send 417
send method 417
send_multiple_requests 289
send_multiple_requests function 289
set_exception 326
set_exception method 326
set_item 378
set_item method 378
set_one_value 337
set_one_value method 337
set_values 339
set_values method 339
SOM kernel
  Macros 92, 96, 98, 100, 101, 103, 105, 112, 114, 117, 119, 121, 123
  SOMClass class 124, 128, 131, 133, 135, 136, 138, 140, 143, 146, 148, 150, 152, 154, 156, 158, 160, 162, 164, 166, 168, 170, 172, 174, 178, 180, 182, 184, 186, 189, 192, 195, 198
    somAddDynamicMethod method 128
    somAllocate method 131
    somCheckVersion method 133
    somClassReady method 135
    somDeallocate method 136

SOM kernel *(continued)*
SOMClass class *(continued)*
somDescendedFrom method 138
somFindMethod(OK) methods 140, 143
somFindSMethod(OK) methods 146, 148
somGetInstancePartSize method 150
somGetInstanceSize method 152
somGetInstanceToken method 154
somGetMemberToken method 156
somGetMethodData method 158
somGetMethodDescriptor method 160
somGetMethodIndex method 162
somGetMethodToken method 164
somGetName method 166
somGetNthMethodData method 168
somGetNthMethodInfo 170
somGetNumMethods method 172
somGetNumStaticMethods method 174
somGetVersionNumbers method 178
somLookupMethod method 180
somNew(NoInit) methods 182, 184
somRenew(NoInit) methods 186, 189, 192, 195
somSupportsMethod method 198
SOMClassMgr class 200, 202, 204, 207, 210, 212, 214, 216, 218, 221, 222, 224, 226
SOMObject class 228, 246, 250, 255, 256, 259, 261, 263, 265, 267, 270, 272, 274, 278, 280
SOM_Assert 92
SOM_Assert macro 92
SOM_ClassLibrary 94
SOM_CreateLocalEnvironment Macro 96
SOM_CreateLocalEnvironment macro 96
SOM_DestroyLocalEnvironment 98
SOM_DestroyLocalEnvironment macro 98
SOM_Error 100
SOM_Error macro 100
SOM_Expect 101
SOM_Expect macro 101
SOM_GetClass 103
SOM_GetClass macro 103
SOM_InitEnvironment 105
SOM_InitEnvironment macro 105
SOM_MainProgram 107
SOM_NoTrace 109
SOM_ParentNumResolve 110
SOM_Resolve 112
SOM_Resolve macro 112
SOM_ResolveNoCheck 114
SOM_ResolveNoCheck macro 114
SOM_SubstituteClass 116
SOM_Test 117
SOM_Test macro 117
SOM_TestC 119
SOM_TestC macro 119
SOM_UninitEnvironment 121
SOM_UninitEnvironment macro 121
SOM_WarnMsg 123
SOM_WarnMsg macro 123
somAddDynamicMethod 128
somAddDynamicMethod method 128
somAllocate 131
somAllocate method 131
somApply 3
somBuildClass 6
SOMCalloc 74
SOMCalloc function 74
somCastObj 230
somCheckId 8
somCheckId function 8
somCheckVersion 133
somCheckVersion method 133
SOMClass class 124
somClassDispatch 250
somClassDispatch method 246, 250
somClassFromId 202
somClassFromId method 202
somClassInitFuncName 76
SOMClassInitFuncName function 76
SOMClassMgr class 200
somClassReady 135
somClassReady method 135
somClassResolve 9
somCompareIds 12
SOMD_Init 293
SOMD_Init function 293
SOMD_NoORBfree 295
SOMD_RegisterCallback 297
SOMD_RegisterCallback function 297
SOMD_Uninit 299
SOMD_Uninit function 299
somDataResolve 14
somDataResolve function 14
somDataResolveChk 16
somDataResolveChk function 16
somdCreateObj 472
somdCreateObj method 472
somdDeleteObj 474
somdDeleteObj method 474

somdDestroyObject 383
somdDestroyObject method 383
somdDisableServer 487
somdDispatchMethod 476
somdDispatchMethod method 476
somDeallocate 136
somDeallocate method 136
somDefaultAssign 232
somDefaultConstAssign 234
somDefaultConstCopyInit 236
somDefaultCopyInit 238
somDefaultInit 240
SOMDeleteModule 78
SOMDeleteModule function 78
somdEnableServer 489
somDescendedFrom 138
somDescendedFrom method 138
somDestruct 243
somdExceptionFree 291
somdFindAnyServerByClass 463
somdFindAnyServerByClass method 463
somdFindServer 465
somdFindServer method 465
somdFindServerByName 467
somdFindServerByName method 467
somdFindServersByClass 469
somdFindServersByClass method 469
somdGetClassObj 478
somdGetClassObj method 478
somdGetIdFromObject 385
somdGetIdFromObject method 385
somdGetObjectFromId 387
somdGetObjectFromId method 387
somDispatch 246
somDispatch method 246, 250
somdIsServerEnabled 491
somdListServer 493
somdNewObject 389
somdNewObject method 389
SOMDObjectMgr class 461
somdObjReferencesCached 480
somdProxyFree 421
somdProxyFree method 421
somdProxyGetClass 423
somdProxyGetClass method 423
somdProxyGetClassName 425
somdProxyGetClassName method 425
somdRefFromSOMObj 482
somdRefFromSOMObj method 482
somdReleaseObject 391
somdReleaseObject method 391
somdReleaseResources 427
somdRestartServer 495
somdShutdownServer 497
somdSOMObjFromRef 484
somdSOMObjFromRef method 484
somdStartServer 499
somdTargetFree 429
somdTargetFree method 429
somdTargetGetClass 431
somdTargetGetClass method 431
somdTargetGetClassName 433
somdTargetGetClassName method 433
somDumpSelf 255
somDumpSelf method 255
somDumpSelfInt 256
somDumpSelfInt method 256
someChangeRegData 608
someChangeRegData method 608
someClearRegData 631
someClearRegData method 631
SOMEClientEvent class 601
SOMEEMan class 606
SOMEEMRegisterData class 630
SOMEEvent class 642
someGetEManSem 610
someGetEManSem method 610
somEnvironmentEnd 18
somEnvironmentNew 19
somEnvironmentNew function 19
someProcessEvent 612
someProcessEvent method 612
someProcessEvents 614
someProcessEvents method 614
someQueueEvent 616
someQueueEvent method 616
someRegister 618
someRegister method 618
someRegisterEv 620
someRegisterEV method 620
someRegisterProc 622
someRegisterProc method 622
someReleaseEManSem 624
someReleaseEManSem method 624
SOMError 79
SOMError function 79
someSetRegDataClientType 632
someSetRegDataClientType method 632

someSetRegDataEventMask 633
someSetRegDataEventMask method 633
someSetRegDataSink 635
someSetRegDataSink method 635
someSetRegDataSinkMask 636
someSetRegDataSinkMask method 636
someSetRegDataTimerCount 638
someSetRegDataTimerCount method 638
someSetRegDataTimerInterval 640
someSetRegDataTimerInterval method 640
someShutdown 626
someShutdown method 626
SOMESinkEvent class 647
SOMETimerEvent class 650
someUnRegister 628
someUnRegister method 628
somevGetEventClientData 602
somevGetEventClientType 603
somevGetEventClientType method 603
somevGetEventInterval 651
somevGetEventInterval method 651
somevGetEventSink 648
somevGetEventSink method 648
somevGetEventTime 643
somevGetEventTime method 643
somevGetEventType 644
somevGetEventType method 644
somevSetEventClientData 604
somevSetEventClientType 605
somevSetEventClientType method 605
somevSetEventInterval 652
somevSetEventInterval method 652
somevSetEventSink 649
somevSetEventSink method 649
somevSetEventTime 645
somevSetEventTime method 645
somevSetEventType 646
somevSetEventType method 646
SOMEWorkProcEvent class 653
somExceptionFree 21
somExceptionFree function 21
somExceptionId 23
somExceptionId function 23
somExceptionValue 25
somExceptionValue function 25
somFindClass 204
somFindClass method 204
somFindClsInFile 207
somFindClsInFile method 207
somFindMethod 140
somFindMethod(OK) methods 140, 143
somFindMethodOk 143
somFindSMethod 146
somFindSMethod(OK) methods 146, 148
somFindSMethodOk 148
somFree 81, 259
SOMFree function 81
somFree method 259
somGetClass 261
somGetClass method 261
somGetClassName 263
somGetClassName method 263
somGetGlobalEnvironment 27
somGetGlobalEnvironment function 27
somGetInitFunction 210
somGetInitFunction method 210
somGetInstancePartSize 150
somGetInstancePartSize method 150
somGetInstanceSize 152
somGetInstanceSize method 152
somGetInstanceToken 154
somGetInstanceToken method 154
somGetMemberToken 156
somGetMemberToken method 156
somGetMethodData 158
somGetMethodData method 158
somGetMethodDescriptor 160
somGetMethodDescriptor method 160
somGetMethodIndex 162
somGetMethodIndex method 162
somGetMethodToken 164
somGetMethodToken method 164
somGetname 166
somGetName method 166
somGetNthMethodData 168
somGetNthMethodData method 168
somGetNthMethodInfo 170
somGetNthMethodInfo method 170
somGetNumMethods 172
somGetNumMethods method 172
somGetNumStaticMethods 174
somGetNumStaticMethods method 174
somGetParents 176
somGetRelatedClasses 212
somGetRelatedClasses method 212
somGetSize 265
somGetSize method 265
somGetVersionNumbers 178

somGetVersionNumbers method 178
somIdFromString 29
somIdFromString function 29
somInit 267
somInit method 267
SOMInitModule 83
somIsA 270
somIsA method 270
somIsInstanceOf 272
somIsInstanceOf method 272
somIsObj 30
somIsObj function 30
somLoadClassFile 214
somLoadClassFile method 214
SOMLoadModule 86
SOMLoadModule function 86
somLocateClassFile 216
somLocateClassFile method 216
somLookupMethod 180
somLookupMethod method 180
somLPrintf 32
somLPrintf function 32
sommAfterMethod 588
somMainProgram 34
SOMMalloc 88
SOMMalloc function 88
sommBeforeMethod 591
somMergeInto 218
somMergeInto method 218
sommGetSingleInstance 595
sommGetSingleInstance method 595
SOMMSingleInstance class 594
SOMMTraced Metaclass
somNew 182
somNew(NoInit) methods 182, 184
somNewNoInit 184
SOMOA class 501
SOMObject class 228
SOMOutCharRoutine 89
SOMOutCharRoutine function 89
somParentNumResolve 35
somParentNumResolve function 35
somParentResolve 38
somParentResolve function 38
somPrefixLevel 40
somPrefixLevel function 40
somPrintf 41
somPrintf function 41
somPrintSelf 274
somPrintSelf method 274
SOMRealloc 91
SOMRealloc function 91
somRegisterClass 221
somRegisterClass method 221
somRegisterId 43
somRegisterId function 43
somRenew 186
somRenew(NoInit) methods 186, 189, 192, 195
somRenewNoInit 189
somRenewNoInitNoZero 192
somRenewNoZero 195
somResetObj 276
somResolve 45
somResolve function 45
somResolveByName 48
somResolveByName function 48
somRespondsTo 278
somRespondsTo method 278
somSetException 50
somSetException function 50
somSetExpectedIds 53
somSetExpectedIds function 53
somSetOutChar 55
somStringFromId 56
somStringFromId function 56
somSubstituteClass 222
somSubstituteClass method 222
somSupportsMethod 198
somSupportsMethod method 198
somTotalRegIds 57
somTotalRegIds function 57
somUninit 280
somUninit method 280
somUniqueKey 59
somUniqueKey function 59
somUnloadClassFile 224
somUnloadClassFile method 224
somUnregisterClass 226
somUnregisterClass method 226
somVaBuf_add 61
somVaBuf_add function 61
somVaBuf_create 63
somVaBuf_create function 63
somVaBuf_destroy 66
somVaBuf_destroy function 66
somVaBuf_get_valist 67
somVaBuf_get_valist function 67
somvalistGetTarget 68

somvalistGetTarget function 68
somvalistSetTarget 70
somvalistSetTarget function 70
somVprintf 72
somVprintf function 72
string_to_object 402
string_to_object method 402

T

TypeCode_alignment 560
TypeCode_alignment function 560
TypeCode_copy 561
TypeCode_copy function 561
TypeCode_equal 563
TypeCode_equal function 563
TypeCode_free 565
TypeCode_free function 565
TypeCode_kind 567
TypeCode_kind function 567
TypeCode_param_count 576
TypeCode_param_count function 576
TypeCode_parameter 577
TypeCode_parameter function 577
TypeCode_print 580
TypeCode_print function 580
TypeCode_setAlignment 582
TypeCode_setAlignment function 582
TypeCode_size 583
TypeCode_size function 583
TypeCodeNew 572
TypeCodeNew function 572

U

update_impldef 364
update_impldef method 364

W

within 528
within method 528

Communicating Your Comments to IBM

VisualAge for C++ for Windows
SOM Programming Reference

Version 3.5

Publication No. S33H-5044-00

If there is something you like—or dislike—about this book, please let us know. You can use one of the methods listed below to send your comments to IBM. If you want a reply, include your name, address, and telephone number. If you are communicating electronically, include the book title, publication number, page number, or topic you are commenting on.

The comments you send should only pertain to the information in this book and its presentation. To request additional publications or to ask questions or make comments about the functions of IBM products or systems, you should talk to your IBM representative or to your IBM authorized remarketer.

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

If you are mailing a readers' comment form (RCF) from a country other than the United States, you can give it to the local IBM branch office or IBM representative for postage-paid mailing.

- If you prefer to send comments by mail, use the RCF at the back of this book.
- If you prefer to send comments by FAX, use this number:
  - United States and Canada: 416-448-6161
  - Other countries: (+1)-416-448-6161
- If you prefer to send comments electronically, use the network ID listed below. Be sure to include your entire network address if you wish a reply.
  - Internet: torrcf@vnet.ibm.com
  - IBMLink: toribm(torrcf)
  - IBM/PROFS: torolab4(torrcf)
  - IBMMAIL: ibmmail(caibmwt9)

Readers' Comments — We'd Like to Hear from You

VisualAge for C++ for Windows
SOM Programming Reference

Version 3.5

Publication No. S33H-5044-00

Overall, how satisfied are you with the information in this book?

	Very Satisfied	Satisfied	Neutral	Dissatisfied	Very Dissatisfied
Overall satisfaction	□	□	□	□	□

How satisfied are you that the information in this book is:

	Very Satisfied	Satisfied	Neutral	Dissatisfied	Very Dissatisfied
Accurate	□	□	□	□	□
Complete	□	□	□	□	□
Easy to find	□	□	□	□	□
Easy to understand	□	□	□	□	□
Well organized	□	□	□	□	□
Applicable to your tasks	□	□	□	□	□

Please tell us how we can improve this book:

Thank you for your responses. May we contact you? □ Yes □ No

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

Name

Address

Company or Organization

Phone No.

Readers' Comments — We'd Like to Hear from You
S33H-5044-00

Cut or Fold
Along Line

Fold and Tape **Please do not staple** Fold and Tape

PLACE
POSTAGE
STAMP
HERE

IBM Canada Ltd. Laboratory
Information Development
2G/345/1150/TOR
1150 EGLINTON AVENUE EAST
NORTH YORK ONTARIO CANADA M3C 1H7

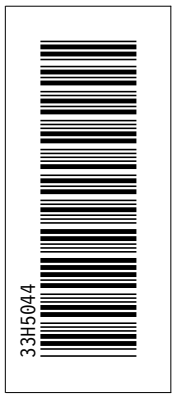
Fold and Tape **Please do not staple** Fold and Tape

S33H-5044-00

Cut or Fold
Along Line

Part Number: 33H5044
Program Number: 33H4979
33H4980

Printed in U.S.A.



S33H-5044-00