SOMobjects Developer's Toolkit

Programmer's Reference, Volume I: SOM and DSOM

SOMobjects Version 3.0

Note: Before using this information and the product it supports, be sure to read the general information under **Notices** on page iii.

Second Edition (December 1996)

This edition of *Programmer's Reference, Volume I: SOM and DSOM* applies to SOMobjects Developer's Toolkit for SOM Version 3.0 and to all subsequent releases of the product until otherwise indicated in new releases or technical newsletters.

To initiate changes to this publication, submit a problem report from the technical support web page at URL: http://www.austin.ibm.com/somservice/supform.html. Otherwise, address comments to IBM Corporation, Internal Zip 1002, 11400 Burnet Road, Austin, Texas 78758-3493. IBM Corporation may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

Requests for copies of this publication and for technical information about IBM products should be made to your IBM Authorized Dealer or your IBM Marketing representative.

Notices

IBM Corporation may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to the IBM Director of Commercial Relations, IBM Corporation, Purchase, NY 10577.

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

Each copy of any portion of these sample programs or any derivative work, which is distributed to others, must include a copyright notice as follows: "© (your company name) (current year), All Rights Reserved." However, the following copyright notice protects this documentation under the Copyright Laws of the United States and other countries which prohibit such actions as, but not limited to, copying, distributing, modifying, and making derivative works.

References in this publication to IBM products, program, or services do not imply that IBM Corporation intends to make these available in all countries in which it operates.

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

IBM Corporation 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, New York 10594, USA

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
Department 931S
11400 Burnet Road
Austin, Texas 78758 USA

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

Asia-Pacific users can inquire, in writing, to the:

IBM Director of Intellectual Property and Licensing
IBM World Trade Asia Corporation,
2-31 Roppongi 3-chome,
Minato-ku, Tokyo 106, Japan

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.

Trademarks and Acknowledgements

AIX is a trademark of International Business Machines Corporation.
FrameViewer is a trademark of Frame Technology.
IBM is a registered trademark of International Business Machines Corporation.
OS/2 is a trademark of International Business Machines Corporation.
SOM is a trademark of International Business Machines Corporation.
SOMobject is a trademark of International Business Machines Corporation.
Windows and WIndows NT are trademarks of Microsoft Corporation.

Table of Contents

Chapter 1. SOM Kernel

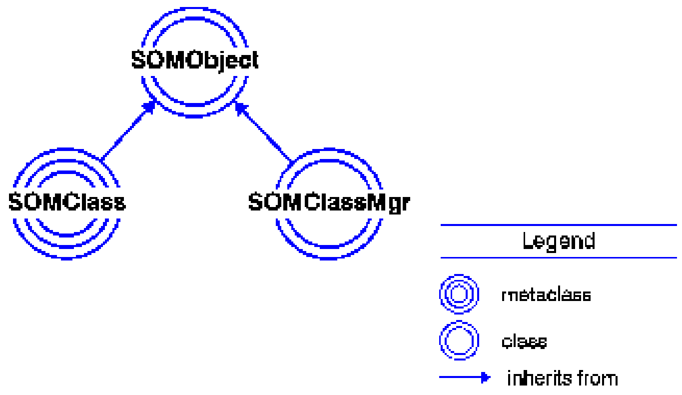

Figure 1. SOM Kernel Class Organization

somApply Function

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

C Syntax

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

Description

somApply provides a single uniform interface through which you can 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 **somApply** 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 **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. There may be methods for which **somApply** cannot select an appropriate apply stub.

Parameters

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

retVal
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
A pointer to the **somMethodData** structure that describes the method whose procedure is to be executed by the apply stub.

args
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.

Return Value

The **somApply** function returns 1 (TRUE) if it executes successfully, or 0 (FALSE) otherwise.

C++ Example

```
#include <somcls.h>
#include <string.h>
#include <stdarg.h>
main()
{
   somVaBuf vb;
   va_list args;
   string result;
   SOMClas *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” */
}
```

Related Information

somGetMethodData Method

somGetNthMethodData Method

somAddDynamicMethod Method (somcls.idl)

SOMObject (somobj.idl)

somMethodData (somapi.h)

somToken (sombtype.h)

somMethodPtr (sombtype.h)

va_list (stdarg.h)

somBeginPersistentIds Function

Tells SOM to begin a “persistent ID interval.”

C Syntax

void somBeginPersistentIds ( );

Description

The **somBeginPersistentIds** function informs the SOM ID manager that strings for any new SOM IDs that are registered will not be freed or modified. This allows the ID manager to use a pointer to the string in the unregistered ID as the master copy of the ID’s string, rather than making a copy of the string. This makes ID handling more efficient.

C Examples

```
#include <som.h>
/* This is the way to create somIds efficiently */
static string id1Name = ”whoami”;
static somId somId_id1 = &id1Name;
/*
   somId_id1 will be registered the first time it is usde in an
   operation that takes a somId, or it can be explicitely registered
   using somCheckId.
*/
main()
{
   somId id1, id2;
   string id2Name = ”whereami”;
   somEnvironmentNew();
   somBeginPersistentIds();
   id1 = somCheckId(somId_id1);    /* registers id as persistent */
   somEndPersistentIds();
   id2 = somIdFromString(id2Name); /* registers the id */
   SOM_Assert(!strcmp(”whoami”, somStringFromId(id1)), SOM_Fatal);
   SOM_Assert(!strcmp(”whereami”, somStringFromId(id2)), SOM_Fatal);
   id1Name = ”it does matter”;    /* is persistent     */
   id2Name = ”it doesn’t matter”; /* is not persistent */
   SOM_Assert(strcmp(”whoami”, somStringFromId(id1)), SOM_Fatal);
                                  /* The id1 string has changed */
   SOM_Assert(!strcmp(”whereami”, somStringFromId(id2)), SOM_Fatal);
                                  /* the id2 string has not     */
}
```

Related Information

somCheckId Function

somCompareIds Function

somEndPersistentIds Function

somIdFromString Function

somRegisterId Function

somSetExpectedIds Function

somStringFromId Function

somTotalRegIds Function

somUniqueKey Function

somBuildClass Function

Automates the process of building a new SOM class object.

C Syntax

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

Description

The **somBuildClass** 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.

Parameters

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

sciPtr
A pointer to a structure holding static class information.

majorVersion
The major version number for the class.

minorVersion
The minor version number for the class.

Example

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

Return Value

The **somBuildClass** function returns a pointer to a class object.

Related Information

somStaticClassInfo (somapi.h)

somCheckId Function

Registers a SOM ID.

C Syntax

somId somCheckId (somId *id*);

Description

The **somCheckId** 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.

Parameters

id
The **somId** to be registered.

Return Value

The registered **somId.**

Example

See **somBeginPersistentIds Function** on page 4.

Related Information

somBeginPersistentIds Function

somCompareIds Function

somEndPersistentIds Function

somIdFromString Function

somRegisterId Function

somSetExpectedIds Function

somStringFromId Function

somTotalRegIds Function

somUniqueKey Function 0

somId (sombtype.h)

somClassResolve Function

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

C Syntax

somMethodPtr somClassResolve (SOMClass *clsPtr,* somMToken *mToken*);

Description

The **somClassResolve** 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. **somClassResolve** 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.

Parameters

clsPtr
A pointer to the class object whose instance method procedure is required.

mToken
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` (that is, at **XYZClassData.foo**). Method tokens can also be obtained using the **somGetMethodToken** method.

Return Value

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

C++ Example

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

Related Information

somParentResolve Function

somParentNumResolve Function

somResolve Function

somResolveByName Function

somClassDispatch Method

somFindMethod(Ok) Methods

somGetMethodToken Method

SOM_Resolve Macros

SOM_ResolveNoCheck Macro

somMethodPtr (sombtype.h)

SOMClass (somcls.idl)

somMToken (somapi.h)

somCompareIds Function

Determines whether two SOM IDs represent the same string.

C Syntax

int somCompareIds (somId *id1,* somId *id2*);

Description

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

Parameters

id1
The first SOM ID to be compared.

id2
The second SOM ID to be compared.

Return Value

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

C Example

```
#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));
}
```

Related Information

somBeginPersistentIds Function

somCheckId Function

somEndPersistentIds Function

somIdFromString Function

somRegisterId Function

somSetExpectedIds Function

somStringFromId Function

somTotalRegIds Function

somUniqueKey Function

somId (sombtype.h)

somDataResolve Function

Accesses instance data within an object.

C Syntax

somToken somDataResolve (SOMObject *obj,* somDToken *dToken*);

Description

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 *className***GetData** macro (which expands to a usage of **somDataResolve**).

Parameters

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

dToken
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.

Return Value

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 run-time error and terminates execution.

Example

The following C and 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);
```

Related Information

somToken (somapi.h)

SOMObject (somobj.idl)

somDToken (sombtype.h)

somDataResolveChk Function

Accesses instance data within an object.

C Syntax

somToken somDataResolveChk (SOMObject *obj*, somDToken *dToken*);

Description

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 *className***GetData** macro (which expands to a usage of **somDataResolve**).

Parameters

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

dToken
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.

Return Value

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.

Example

The following C and 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);
```

Related Information

somToken (somapi.h)

SOMObject (somobj.idl)

somDToken (sombtype.h)

somEndPersistentIds Function

Tells SOM to end a persistent ID interval.

C Syntax

void somEndPersistentIds ( );

Description

The **somEndPersistentIds** function informs the SOM ID manager that strings for any new SOM IDs that are registered might be freed or modified by the client program. Thus, the ID manager must make a copy of the strings.

Example

See **somBeginPersistentIds Function** on page 4.

Related Information

somBeginPersistentIds Function

somCheckId Function

somCompareIds Function

somIdFromString Function

somRegisterId Function

somSetExpectedIds Function

somStringFromId Function

somTotalRegIds Function

somUniqueKey Function

somEnvironmentNew Function

Initializes the SOM runtime environment.

C Syntax

SOMClassMgr somEnvironmentNew ( );

Description

The **somEnvironmentNew** function creates the four primitive SOM objects (**SOMObject**, **SOMClass**, **SOMClassMgr** 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 *className***New** macros, the *className***Renew** macros, the **new** operator, and the *className***NewClass** procedures defined by the SOM C and C++ language bindings call **somEnvironmentNew** if needed.

Return Value

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

Example

```
somEnvironmentNew();
```

Related Information

somExceptionId Function

somExceptionValue Function

somGetGlobalEnvironment Function

somSetException Function

somExceptionFree Function

Frees the memory held by the exception structure within an **Environment** structure.

C Syntax

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

Description

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

Parameters

ev
A pointer to the **Environment** whose exception information is to be freed.

Example

See somSetException Function on page 32.

Related Information

somdExceptionFree Function

somExceptionId Function

somExceptionValue Function

somGetGlobalEnvironment Function

somSetException Function

Environment (somcorba.h)

somExceptionId Function

Gets the name of the exception contained in an **Environment** structure.

C Syntax

string somExceptionId (Environment **ev*);

Description

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

Parameters

ev
A pointer to an **Environment** structure containing an exception.

Return Value

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

Example

See **somSetException Function** on page 32.

Related Information

somExceptionFree Function

somExceptionValue Function

somGetGlobalEnvironment Function

somSetException Function

string (somcorba.h)

Environment (somcorba.h)

somExceptionValue Function

Gets the value of the exception contained in an **Environment** structure.

C Syntax

somToken somExceptionValue (Environment **ev*);

Description

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

Parameters

ev
A pointer to an **Environment** structure containing an exception.

Return Value

The **somExceptionValue** function returns a pointer to the value of the exception contained in the specified **Environment** structure.

Example

See **somSetException Function** on page 32.

Related Information

somExceptionFree Function

somExceptionId Function

somGetGlobalEnvironment Function

somSetException Function

somToken (sombtype.h)

Environment (somcorba.h)

somGetGlobalEnvironment Function

Returns a pointer to the current global **Environment** structure.

C Syntax

Environment *somGetGlobalEnvironment ( );

Description

The **somGetGlobalEnvironment** 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->_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** functions.

Return Value

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

Example

See **somSetException Function** on page 32.

Related Information

somExceptionId Function

somExceptionValue Function

somExceptionFree Function

somSetException Function

Environment (somcorba.h)

somIdFromString Function

Returns the SOM ID corresponding to a given text string.

C Syntax

somId somIdFromString (string *aString);*

Description

The **somIdFromString** function returns the SOM ID that corresponds to a given text string.

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

Parameters

aString
The string to be converted to a SOM ID.

Return Value

Returns the SOM ID corresponding to the given text string.

Example

See **somBeginPersistentIds Function** on page 4.

Related Information

somBeginPersistentIds Function

somCheckId Function

somCompareIds Function

somEndPersistentIds Function

somRegisterId Function

somSetExpectedIds Function

somStringFromId Function

somTotalRegIds Function

somUniqueKey Function

somId (sombtype.h)

string (somcorba.h)

somIsObj Function

Failsafe routine to determine whether a pointer references a valid SOM object.

C Syntax

boolean somIsObj (somToken *memPtr);*

Description

The **somIsOb**j 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.

Parameters

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

Return Value

The **somIsObj** function returns 1 if *obj* is a pointer to a valid SOM object, and 0 otherwise.

C++ Example

```
#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());
}
```

Related Information

boolean (somcorba.h)

somToken (sombtype.h)

somLPrintf Function

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

C Syntax

long somLPrintf (long *level,* string *fmt,* ...);

Description

The **somLPrintf** 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.

Parameters

level
The level at which output is to be placed.

fmt
The format string to be output.

varargs
The values to be substituted into the format string.

Return Value

Returns the number of characters written.

C Example

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

Related Information

somPrefixLevel Function

somPrintf Function

somVprintf Function

SOMOutCharRoutine Function

string (somcorba.h)

somMainProgram Function

Performs SOM initialization on behalf of a new program.

C Syntax

SOMClassMgr *somMainProgram ( );

Description

The **somMainProgram** function informs SOM about the beginning of a new thread of execution. The SOM Kernel then performs any needed initialization, including the deferred execution of the **SOMInitModule Functions** found in statically-loaded class libraries. When **somMainProgram** is used, it supersedes any need to call the **somEnvironmentNew** function.

Return Value

A pointer to the **SOMClassMgr Class** object.

Related Information

somEnvironmentNew Function

somParentNumResolve Function

Obtains a pointer to a procedure that implements a method, given a list of method tables.

C Syntax

somMethodPtr somParentNumResolve (
somMethodTabs ***parentMtab,***
int ***parentNum*,**
somMToken ***mToken);***

Description

The **somParentNumResolve** 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).

Parameters

parentMtab
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 **XYZCClassData.parentMtab**.

parentNum
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
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` (that is, at **XYZClassData.foo**). Method tokens can also be obtained using the **somGetMethodToken** method.

Return Value

A **somMethodPtr** pointer to the procedure that implements the specified method, selected from the specified method table.

C++ Example

```
// SOM IDL for class A and class B
#include <somobj.idl>
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) // foo method token
      ) /* end of method procedure expression */
      (objPtr); /* method arguments */
}
```

Related Information

somClassResolve Function

somParentNumResolve Function

somResolve Function

somResolveByName Function

somGetMethodToken Method

SOM_ParentNumResolve Macro

SOM_Resolve Macro

SOM_ResolveNoCheck Macro

somMethodPtr (sombtype.h)

somMethodTabs (somapi.h)

somMToken (somapi.h)

somParentResolve Function

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

C Syntax

```
somMethodPtr  somParentResolve (
                    somMethodTabs parentMtab,
                    somMToken mToken);
```

Description

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

Parameters

parentMtab
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**.

mToken
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` (that is, at **XYZClassData.foo**). Method tokens can also be obtained using the **somGetMethodToken** method.

Return Value

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

Related Information

somClassResolve Function

somParentResolve Function

somResolve Function

somResolveByName Function

somClassDispatch Method

somFindMethod(Ok) Methods

somGetMethodToken Method

SOM_Resolve Macro

SOM_ResolveNoCheck Macro

somMethodPtr (sombtype.h)

somMethodTabs (somapi.h)

somMToken (somapi.h).

somPrefixLevel Function

Outputs blanks to prefix a line at the indicated level.

C Syntax

void somPrefixLevel (long *level*);

Description

The **somPrefixLevel** function outputs blanks via **somPrintf** to prefix the next line of output at the indicated level. The number of blanks produced is 2*level.

This function is useful when overriding the **somDumpSelfInt Method**.

Parameters

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

C/C++ Example

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

Related Information

somLPrintf Function

somPrintf Function

somVprintf Function

SOMOutCharRoutine Function

somPrintf Function

Prints a formatted string in the manner of the C **printf** function.

C Syntax

long somPrintf (string *fmt,* ...);

Description

The **somPrintf** function prints a formatted string using the **SOMOutCharRoutine** function, 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.

Note: If you use calls to both **somPrintf** and **printf**, you should be aware that these functions use different buffers and that the output may not appear in the same order as the calls occur in the code.

Parameters

fmt
The format string to be output.

varargs
The values to be substituted into the format string.

Return Value

Returns the number of characters written.

C Example

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

Related Information

somLPrintf Function

somPrefixLevel Function

somVprintf Function

SOMOutCharRoutine Function

somRegisterId Function

Registers a SOM ID and determines whether or not it was previously registered.

C Syntax

int somRegisterId (somId *id);*

Description

somRegisterId registers a SOM ID and converts it into an internal representation.

Parameters

id
The **somId** to be registered.

Return Value

If the ID is registered, returns 0; otherwise, returns 1.

C Example

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

Related Information

somBeginPersistentIds Function

somCheckId Function

somCompareIds Function

somEndPersistentIds Function

somIdFromString Function

somSetExpectedIds Function

somStringFromId Function

somTotalRegIds Function

somUniqueKey Function

somId (sombtype.h).

somResolve Function

Obtains a pointer to the procedure that implements a method for a particular SOM object.

C Syntax

somMethodPtr somResolve (SOMObject *obj,* somMToken *mToken);*

Description

The **somResolve** 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**.

Parameters

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

mToken
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` (that is, at **XYZClassData.foo**). Method tokens can also be obtained using the **somGetMethodToken** method.

Return Value

A **somMethodPtr** pointer to the procedure that implements the specified method for the specified SOM object.

C Example

```
// 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);
}
```

Related Information

somClassResolve Function

somParentNumResolve Function

somParentResolve Function

somResolveByName Function

somClassDispatch Method

somFindMethod(Ok) Methods

somGetMethodToken Method

SOM_Resolve Macro

SOM_ResolveNoCheck Macro

somMethodPtr (sombtype.h)

somMToken (somapi.h).

somResolveByName Function

Obtains a pointer to the procedure that implements a method for a particular SOM object.

C Syntax

somMethodPtr somResolveByName (SOMObject *obj*, string *methodName*);

Description

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

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

Parameters

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

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

Return Value

A **somMethodPtr** pointer to the procedure that implements the method for the SOM object.

C Example

Assuming the static method `setSound` is introduced by the class `Animal`, this example will invoke this method on an instance of `Animal` or a descendent class.

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

Related Information

somResolve Function

somParentResolve Function

somParentNumResolve Function

somClassResolve Function

somClassDispatch Method

somFindMethod(Ok) Methods

SOM_Resolve Macro

SOM_ResolveNoCheck Macro

somMethodPtr (sombtype.h)

SOMObject (somobj.idl)

string (somcorba.h)

somSetException Function

Sets an exception value in an **Environment** structure.

C Syntax

void somSetException (
Environment *_ev,_
enum exception_type _major,_
string _exceptionName,_
somToken _params_);

Description

The **somSetException** function sets an exception value in an **Environment** structure.

Parameters

ev
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
An integer representing the type of exception to set.

exceptionName
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 exception name. Where unambiguous, the usage bindings also define the short form `ex_`*`exceptionName`*, where *`exceptionName`* is unqualified.

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

C Example

```
/* 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 copied id and original X_OUCH structure: */
        somExceptionFree(ev);
   }
}
```

Related Information

somExceptionFree Function

somExceptionId Function

somExceptionValue Function

somGetGlobalEnvironment Function

Environment

exception_type

string (somcorba.h)

somSetExpectedIds Function

Tells SOM how many unique SOM IDs a client program expects to use.

C Syntax

void somSetExpectedIds (unsigned long *numIds);*

Description

The **somSetExpectedIds** 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.

Parameters

numIds
The number of SOM IDs the client program expects to use.

C Example

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

Related Information

somBeginPersistentIds Function

somCheckId Function

somCompareIds Function

somEndPersistentIds Function

somIdFromString Function

somRegisterId Function

somStringFromId Function

somTotalRegIds Function

somUniqueKey Function

somSetOutChar Function

Changes the behavior of the **somPrintf** function.

C Syntax

void somSetOutChar (somTD_SOMOutCharRoutine * *outCharRtn*);

Description

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

Some additional samples of **somSetOutChar** can be found in the **somapi.h** header file.

Parameters

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

Example

```
#include <som.h>
static int irOutChar(char c);
static int irOutChar(char c)
{
    (Customized code goes here.)
}
main (...)
{
    somSetOutChar((somTD_SOMOutCharRoutine *) irOutChar);
}
```

Related Information

somPrintf Function

SOMOutCharRoutine Function

somStringFromId Function

Returns the string that a SOM ID represents.

C Syntax

string somStringFromId (somId *id);*

Description

The **somStringFromId** function returns the string that a given SOM ID represents.

Parameters

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

Return Value

Returns the string that the given SOM ID represents.

Example

See **somBeginPersistentIds Function** on page 4.

Related Information

somBeginPersistentIds Function

somCheckId Function

somCompareIds Function

somEndPersistentIds Function

somIdFromString Function

somRegisterId Function

somSetExpectedIds Function

somTotalRegIds Function

somUniqueKey Function

string (somcorba.h)

somId (sombtype.h)

somTotalRegIds Function

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

C Syntax

unsigned long somTotalRegIds ( );

Description

The **somTotalRegIds** function returns the total number of registered SOM IDs. 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.

Return Value

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

C Example

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

Related Information

somBeginPersistentIds Function

somCheckId Function

somCompareIds Function

somEndPersistentIds Function

somIdFromString Function

somRegisterId Function

somSetExpectedIds Function

somStringFromId Function

somUniqueKey Function

somUniqueKey Function

Returns the unique key associated with a SOM ID.

C Syntax

unsigned long somUniqueKey (somID ***id);***

Description

The **somUniqueKey** 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.

Parameters

id
The SOM ID for which the unique key is needed.

Return Value

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

C Example

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

Related Information

somBeginPersistentIds Function

somCheckId Function

somCompareIds Function

somEndPersistentIds Function

somIdFromString Function

somRegisterId Function

somSetExpectedIds Function

somStringFromId Function

somTotalRegIds Function

somId (**sombtype.h**)

somvalistGetTarget Function

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

Note: There is adanger in using this function. This function treats whatever the **va_list** is currently pointing to as an unsigned long regardless of type. The return value from a double, for example, would be the high 4 bytes of the double, and setting it will merely corrupt the high 4 bytes in the existing **va_list**. There is no guarantee that this function will change the scalar value correctly.

C Syntax

unsigned long somvalistGetTarget (va_list *ap*);

Description

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

Parameters

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

Return Value

Scalar value from the **va_list.**

Example

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

Related Information

somvalistSetTarget Function

somVaBuf_add Function

somVaBuf_create Function

somVaBuf_destroy Function

somVaBuf_get_valist Function

somvalistSetTarget Function

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

Note: There is a danger in using this function. This function treats whatever the **va_list** is currently pointing to as an unsigned long regardless of type. The return value from a double, for example, would be the high 4 bytes of the double, and setting it will merely corrupt the high 4 bytes in the existing **va_list**. There is no guarantee that this function will change the scalar value correctly.

C Syntax

unsigned long somvalistSetTarget (va_list *ap*, unsigned long *val*);

Description

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.

Parameters

ap
The **va_list** to modify.

val
Value to set in the first scalar slot.

Example

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

Related Information

somvalistGetTarget Function

somVaBuf_add Function

somVaBuf_create Function

somVaBuf_destroy Function

somVaBuf_get_valist Function

somVaBuf_add Function

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

C Syntax

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

Description

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

Parameters

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

arg
Pointer to the argument to be added to the **va_list**.

type
Argument type (**TCKind**).

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

tk_boolean
tk_char
tk_double
tk_enum
tk_float
tk_long
tk_octet
tk_pointer
tk_short
tk_string
tk_ulong
tk_ushort
tk_Typecode
Return Value

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

Example

See **somVaBuf_create Function** on page 42.

Related Information

somvalistGetTarget Function

somvalistSetTarget Function

somVaBuf_create Function

somVaBuf_destroy Function

somVaBuf_get_valist Function

somVaBuf_create Function

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

C Syntax

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

Description

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

- *size* is less than the size of the **somVaBuf** structure
- *size* is zero
- *vb* is NULL

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.

Parameters

vb
Pointer to user-allocated memory or NULL

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

Return Value

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

C Example

```
#include <somobj.h>
#include <somtc.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 _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 */
     /* 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>
#include <somtc.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);
}
```

Related Information

somvalistGetTarget Function

somvalistSetTarget Function

somVaBuf_add Function

somVaBuf_destroy Function

somVaBuf_get_valist Function

somVaBuf_destroy Function

Purpose

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

C Syntax

void somVaBuf_destroy (somVaBuf *vb*);

Description

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

Parameters

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

Example

See **somVaBuf_create Function** on page 42.

Related Information

somvalistGetTarget Function

somvalistSetTarget Function

somVaBuf_add Function

somVaBuf_create Function

somVaBuf_get_valist Function

somVaBuf_get_valist Function

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

C Syntax

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

Description

This function assigns a pointer to the passed **va_list** in the **somVaBuf** structure. The caller should not free **va_list**.

Parameters

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

ap
Pointer to a **va_list**.

Example

See **somVaBuf_create Function** on page 42.

Related Information

somvalistGetTarget Function

somvalistSetTarget Function

somVaBuf_add Function

somVaBuf_create Function

somVaBuf_destroy Function

somVprintf Function

Prints a formatted string in the manner of the C **vprintf** function.

C Syntax

long somVprintf (string *fmt*, va_list *ap*);

Description

The **somVprintf** 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.

Parameters

fmt
The format string to be output.

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

Return Value

Returns the number of characters written.

C Example

```
#include <som.h>
#include <somtc.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);
}
```

Related Information

somLPrintf Function

somPrefixLevel Function

somPrintf Function

SOMOutCharRoutine Function

string (somcorba.h)

va_list (stdarg.h)

SOMCalloc Function

Allocates sufficient zeroed memory for an array of objects of a specified size.

C Syntax

somToken (*SOMCalloc) (size_t *num*, size_t *size*);

Description

The **SOMCalloc** function allocates an amount of memory equal to *num***size* (sufficient memory for an array of *num* objects of size *size).* The **SOMCalloc** function has the same interface as the C **calloc** function and performs the same basic function but 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.**

Parameters

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

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

Return Value

A pointer to the first byte of the allocated space.

Example

See **somVprintf Function** on page 46.

Related Information

SOMFree Function

SOMMalloc Function

SOMRealloc Function

somToken (sombtype.h)

SOMClassInitFuncName Function

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

C Syntax

string (*SOMClassInitFuncName) ( );

Description

The **SOMClassInitFuncName** 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**.

Return Value

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

C Example

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

Related Information

SOMDeleteModule Function

SOMLoadModule Function

string (somcorba.h)

SOMDeleteModule Function

Unloads a dynamically linked library (DLL).

C Syntax

int (*SOMDeleteModule) (somToken *modHandle*);

Description

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

Parameters

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

Return Value

Returns 0 if successful or a non-zero system-specific error code otherwise.

Related Information

SOMClassInitFuncName Function

SOMLoadModule Function

somToken (sombtype.h)

SOMError Function

Handles an error condition.

C Syntax

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

Description

The **SOMError** 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)
- `SOM_Ignore` (1)

The default implementation of **SOMError** prints a message that includes the specified error code, file name 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 the **SOMError** function and supplies the last two arguments.

Parameters

errorCode
An integer representing the error code of the error.

fileName
The name of the file in which the error occurred.

lineNum
The line number where the error occurred.

Related Information

SOM_Assert Macro

SOM_Error Macro

SOM_Expect Macro

SOM_Test Macro

SOM_TestC Macro

SOM_WarnMsg Macro

SOMFree Function

Frees the specified block of memory.

C Syntax

void (*SOMFree) (somToken *ptr*);

Description

SOMFree frees the block of memory pointed to by *ptr.* **SOMFree** should only be called with a pointer previously allocated by **SOMMalloc** or **SOMCalloc. SOMFree** has the same interface as the C **free** function and performs the same basic function, but 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 **somFree**, rather than this function.

Parameters

ptr
A pointer to the block of storage to be freed.

C Example

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

Related Information

SOMCalloc Function

SOMMalloc Function

SOMRealloc Function

somFree Method

SOMInitModule Function

Invokes the class creation routines for the classes contained in a class library (DLL).

C Syntax

```
SOMEXTERN void SOMLINK SOMInitModule (
                    long majorVersion,
                    long minorVersion,
                    string className);
```

Description

A class library (DLL) can contain the implementations for multiple classes, all of which should be created when the DLL is loaded. 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. A default class initialization function is generated by the **imod** emitter.

Parameters

majorVersion
The major version number of the class requested when library was loaded.

minorVersion
The minor version number of the class requested when library was loaded.

className
The name of the class requested when library was loaded.

Related Information

SOMClassInitFuncName Function

somGetInitFunction Method

SOMLoadModule Function

Loads the dynamically linked library (DLL) containing a SOM class.

C Syntax

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

Description

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

Parameters

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

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

functionName
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 Function,** obtained from the **SOMClassInitFuncName** function. If no **SOMInitModule** entry exists in the DLL, the default version of **SOMLoadModule** looks for a routine named *className***NewClass** instead. If neither entry point is found, the default version of **SOMLoadModule** fails.

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

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

modHandle
The address where **SOMLoadModule** should place a token that can be subsequently used by the **SOMDeleteModule** routine to unload the DLL.

Return Value

Returns 0 if successful or a non-zero system-specific error code otherwise.

Related Information

SOMClassInitFuncName Function

SOMDeleteModule Function

SOMMalloc Function

Allocates the specified amount of memory.

C Syntax

somToken (*SOMMalloc) (size_t *size*);

Description

The **SOMMalloc** function allocates *size* bytes of memory. The **SOMMalloc** 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.**

Parameters

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

Return Value

A pointer to the first byte of the allocated space.

Example

See **SOMFree Function** on page 51.

Related Information

SOMCalloc Function

SOMFree Function

SOMRealloc Function

SOMOutCharRoutine Function

Prints a character. This function is replaceable.

C Syntax

int (*SOMOutCharRoutine) (char *c*);

Description

SOMOutCharRoutine 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**.

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

Parameters

c
The character to be output.

Return Value

Returns 0 if an error occurs and 1 otherwise.

Example

```
#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;
```

Related Information

somLPrintf Function

somPrefixLevel Function

somPrintf Function

somSetOutChar Function

somVprintf Function

SOMRealloc Function

Changes the size of a previously allocated region of memory.

C Syntax

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

Description

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 and performs the same basic function but 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.**

Parameters

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

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

Return Value

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.)

Related Information

SOMCalloc Function

SOMFree Function

SOMMalloc Function

SOM_Assert Macro

Asserts that a **boolean** condition is true.

Syntax

void **SOM_Assert *(***
boolean *condition,*
long *errorCode*);

Description

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** global variable 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** global variable 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   */
```

Parameters

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

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

Example

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

Related Information

SOM_Expect Macro

SOM_Test Macro

SOM_TestC Macro

SOM_CreateLocalEnvironment Macro

Creates and initializes a local **Environment** structure.

Syntax

Environment * SOM_CreateLocalEnvironment ( );

Description

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.

The **SOM_InitEnvironment** macro differs from the **SOM_CreateLocalEnvironment** macro in how the local Environment structure is created. If the local **Environment** structure is to be created on the stack, use the **SOM_InitEnvironment** macro to initialize it. If the local **Environment** structure is to be created on the heap, use the **SOM_CreateLocalEnvironmen**t macro to create and initialize it.

Expansion

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

C Example

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

Related Information

SOM_DestroyLocalEnvironment Macro

SOM_InitEnvironment Macro

SOM_UninitEnvironment Macro

somGetGlobalEnvironment Function

Environment (somcorba.h)

SOM_DestroyLocalEnvironment Macro

Destroys a local **Environment** structure.

Syntax

SOM_DestroyLocalEnvironment (Environment * *ev*);

Description

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

Parameters

ev
A pointer to the **Environment** structure to be discarded.

Expansion

SOM_DestroyLocalEnvironment first invokes the **somExceptionFree** function on the **Environment** structure; then it invokes the **SOMFree Function** on it to free the memory it occupies.

Example

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

Related Information

SOM_CreateLocalEnvironment Macro

SOM_UninitEnvironment Macro

somExceptionFree Function

SOM_Error Macro

Reports an error condition.

Syntax

void SOM_Error (long *errorCode*);

Description

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.

Parameters

errorCode
The integer error code for the error to be reported.

Related Information

SOMError Function

SOM_Expect Macro

Asserts that a **boolean** condition is expected to be true.

Syntax

void SOM_Expect (boolean *condition*);

Description

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.

Parameters

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

Example

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

Related Information

SOM_Assert Macro

SOM_Test Macro

SOM_TestC Macro

SOM_GetClass Macro

Returns the class object of which a SOM object is an instance.

Syntax

SOMClass SOM_GetClass (SOMObject *objPtr*);

Description

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.

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.

Parameters

objPtr
A pointer to the object whose class is needed.

C++ Example

```
#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");
}
```

Related Information

somGetClass Method

SOM_InitEnvironment Macro

Initializes a local **Environment** structure.

Syntax

void SOM_InitEnvironment (Environment ****ev*);**

Description

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.

The **SOM_InitEnvironment** macro differs from the **SOM_CreateLocalEnvironment** macro in how the local Environment structure is created. If the local Environment structure is to be created on the stack, use the **SOM_InitEnvironment** macro to initialize it. If the local Environment structure is to be created on the heap, use the **SOM_CreateLocalEnvironmen**t macro to create and initialize it.

Parameters

ev
A pointer to the **Environment** structure to be initialized.

Expansion

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

C Example

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

Related Information

SOM_CreateLocalEnvironment Macro

SOM_DestroyLocalEnvironment Macro

SOM_UninitEnvironment Macro

somGetGlobalEnvironment Function

SOM_NoTrace Macro

Turns off method debugging.

Syntax

SOM_NoTrace (<token> *className,* <token> *methodName*);

Description

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.

Parameters

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

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

Expansion

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

Example

Within an implementation file:

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

SOM_ParentNumResolve Macro

Obtains a pointer to a method procedure from a list of method tables. Used by C and C++ implementation bindings to implement parent method calls.

Syntax

somMethodPtr SOM_ParentNumResolve (
<token> ***introClass***,
long ***parentNum***,
somMethodTabs ***parentMtabs,***
<token> ***methodName*****);**

Description

The **SOM_ParentNumResolve** 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*.

Parameters

introClass
The name of the class that introduces *methodName*. This name should be given as a simple token, rather than a quoted string.

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

parentMtabs
A list of parent method tables that the **CClassData.parentMtab** field points to.

methodName
The name of the method to be resolved. This name should be a simple token, rather than a quoted string.

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.

Example

```
#include <somcls.h>
main()
{
SOMClassMgr cm = somEnvironmentNew();
SOM_ParentNumResolve(SOMObject, 1, somClassCClassData.parentMtab,
                         somDumpSelfInt)
                        (_SOMClass,1);
}
```

Related Information

somParentResolve Function

SOM_Resolve Macro

Obtains a pointer to a static method procedure.

Syntax

somMethodPtr SOM_Resolve (
SOMObject *objPtr,*
<token> *className,*
<token> *methodName*);

Description

SOM_Resolve invokes **somResolve** to obtain a pointer to the static method procedure that implements the specified method for the specified object. This pointer provides for efficient repeated casted invocations on instances of the class of the object on which the resolution is done. You must know the class name that introduces the method and the method name to use this macro. Otherwise, use **somResolveByName, somFindMethod** or **somFindMethodOk**. **SOM_Resolve** can only obtain a method procedure for a static method. Unlike **SOM_ResolveNoCheck**, **SOM_Resolve** performs several consistency checks on the objected pointed to by *objPtr.*

Parameters

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

className
The name of the class that introduces *methodName*. This name should be a simple token, rather than a quoted string.

methodName
The name of the method to be resolved. This name should be a simple token, rather than a quoted string.

Expansion

SOM_Resolve uses the *className* and *methodName* to construct the method token for the specified method, then invokes the **somResolve** function. 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.

Example

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

Related Information

SOM_ResolveNoCheck Macro

somClassResolve Function

somResolve Function

somResolveByName Function

somClassDispatch Method

somFindMethod(Ok) Methods

SOM_ResolveNoCheck Macro

Obtains a pointer to a static method procedure without performing consistency checks.

Syntax

somMethodPtr SOM_ResolveNoCheck (
SOMObject *object,*
<token> *className,*
<token> *methodName*);

Description

SOM_ResolveNoCheck invokes the **somResolve** function to obtain a pointer to the method procedure that implements the specified method for the specified object. Use this pointer 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 **somFindMethod** or **somFindMethodOk**.

SOM_ResolveNoCheck 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.*

Parameters

objPtr
A pointer to the object that the resolved method procedure will be applied.

className
The name of the class that introduces *methodName*. This name should be a simple token, rather than a quoted string.

methodName
The name of the method to be resolved. This name should be a simple token, rather than a quoted string.

Expansion

SOM_ResolveNoCheck uses the *className* and *methodName* to construct an expression whose value is the method token for the specified method, then invokes **somResolve**. 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.

Example

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

Related Information

SOM_Resolve Macro

somClassResolve Function

somResolve Function

somResolveByName Function

somClassDispatch Method

somFindMethod(Ok) Methods

SOM_SubstituteClass Macro

Provides a convenience macro for invoking the **somSubstituteClass** method.

Syntax

long SOM_SubstituteClass (
<token> *oldClass*,
<token> *newClass*);

Description

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

Parameters

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

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

Example

See **somSubstituteClass Method** on page 129.

Related Information

somSubstituteClass Method

SOM_Test Macro

Tests whether a **boolean** condition is true; if not, a fatal error is raised.

Syntax

void SOM_Test (boolean *expression*);

Description

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

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

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

Parameters

expression
The **boolean** expression to test.

External (Global) Data

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

C Example

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

Related Information

SOM_Assert Macro

SOM_Expect Macro

SOM_TestC Macro

SOM_TestC Macro

Tests whether a **boolean** condition is true; if not, a warning message is output.

Syntax

void SOM_TestC (boolean *expression*);

Description

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

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

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

Parameters

expression
The **boolean** expression to test.

External (Global) Data

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

C Example

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

Related Information

SOM_Assert Macro

SOM_Expect Macro

SOM_TestC Macro

SOM_UninitEnvironment Macro

Uninitializes a local **Environment** structure.

Syntax

void SOM_UninitEnvironment (Environment ****ev*);**

Description

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

Parameters

ev
A pointer to the **Environment** structure to be uninitialized.

Expansion

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

C Example

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

Related Information

SOM_DestroyLocalEnvironment Macro

SOM_InitEnvironment Macro

SOM_WarnMsg Macro

Reports a warning message.

Syntax

void SOM_WarnMsg (string *msg*);

Description

If **SOM_WarnLevel** global variable 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.

Parameters

msg
The warning message to be output.

Related Information

SOM_Error Macro

SOMClass Class

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. 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** 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: **somDefaultInit**, **somDestruct** and **somDumpSelfInt**. The new methods introduced in **SOMClass** that are frequently overridden are **somNew**, **somRenew** and **somClassReady**.

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.

File Stem

somcls

Base

SOMObject Class

Metaclass

SOMClass

Note: **SOMClass** is the only class with itself as metaclass.

Ancestor Classes

SOMObject Class

Types

```
typedef sequence <SOMClass> SOMClassSequence;

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

C++ Example

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

```
                    so._buffer[i]->offset);
        }
```

New 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.

New Methods

The SOMClass Class introduces the following groups for methods. These method groups are Instance Creations (Factory), Initialization and Termination, Access, Testing and Dynamic.

Group: Instance Creation

somAllocate Method

somDeallocate Method

somNew(NoInit) Methods

somRenew(NoInitNoZero) Methods

Group: Initialization/Termination

somAddDynamicMethod Method

somClassReady Method

Group: Access

somGetInstancePartSize Method

somGetInstanceSize Method

somGetInstanceToken Method

somGetMemberToken Method

somGetMethodData Method

somGetMethodDescriptor Method

somGetMethodIndex Method

somGetMethodToken Method

somGetName Method

somGetNthMethodData Method

somGetNthMethodInfo Method

somGetNumMethods Method

somGetNumStaticMethods Method

somGetParents Method

somGetVersionNumbers Method

Group: Testing

somCheckVersion Method

somDescendedFrom Method

somSupportsMethod Method

Group: Dynamic

somDefinedMethod Method

somFindMethod(Ok) Methods

somFindSMethod(Ok) Method

somLookupMethod Method

Overridden Methods

somDefaultInit Method

somDestruct Method

somDumpSelfInt Method

Deprecated Methods

Use of the following methods is discouraged:

somAddStaticMethod

somGetApplyStub

somGetClassData

somGetClassMtab

somGetInstanceOffset

somGetMethodOffset

somGetParent

somGetPClsMtab

somGetPClsMtabs

somGetRdStub

somInitClass

somInitMIClass

somOverrideMtab

somOverrideSMethod

somSetClassData

somSetMethodDescriptor

_get_somDirectInitClasses attribute

_set_somDirectInitClasses attribute

For these reasons:

- These methods are used in constructing classes, and this capability is provided by the **somBuildClass Function**. 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. The SOM language bindings always use **somBuildClass**.
- 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.
- Some of these methods are now obsolete. Their use is discouraged.

somAddDynamicMethod 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.

IDL Syntax

void somAddDynamicMethod (
in somId *methodId*,
in somId *methodDescriptor*,
in somMethodPtr *method*,
in somMethodPtr *applyStub*);

Description

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 **somAddDynamicMethod** 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.

Parameters

receiver
A pointer to a SOM class object.

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

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

method
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
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 **somClassDispatch Method**) 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.

C Example

```
/* New dynamic method "newMethod1" for class "XXX" */
static char *somMN_newMethod1 = "newMethod1";
static somId somId_newMethod1 = &somMN_newMethod1;
static char *somDS_newMethod1 = "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 *) newMethod1,     /* method procedure  */
   (somMethodProc *) somAP_newMethod1); /* method apply stub */
}
```

Original Class

SOMClass Class

Related Information

somGetMethodDescriptor Method

somAllocate Method

Supports class-specific memory allocation for class instances. Cannot be overridden.

IDL Syntax

string somAllocate (in long *size*);

Description

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.

Parameters

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

size
The number of bytes to be allocated.

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

C++ Example

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

Original Class

SOMClass Class

Related Information

somDeallocate Method

somCheckVersion Method

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

IDL Syntax

boolean somCheckVersion (
In long *majorVersion*,
In long *minorVersion*);

Description

somCheckVersion 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 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.

Parameters

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

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

minorVersion
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.

Return Value

Returns 1 (true) if the implementation of this class is compatible with the specified major and minor version number, and 0 (false) otherwise.

C Example

```
#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
```

Original Class

SOMClass Class

somClassReady Method

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

IDL Syntax

void somClassReady ( );

Description

somClassReady is invoked 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 **somClassReady**. 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 the **somClassReady** method directly; it is invoked automatically during class construction.

Parameters

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

Original Class

SOMClass Class

somDeallocate Method

Frees memory originally allocated by the **somAllocate** method from the same class object. Cannot be overridden.

IDL Syntax

void somDeallocate (in string *memPtr*);

Description

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 registered 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.

Parameters

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

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

Original Class

SOMClass Class

Related Information

somAllocate Method

somDefinedMethod Method

Determines whether a class defines an implementation for a method.

IDL Syntax

somMethodPtr somDefinedMethod (in somMToken *method*);

Note: This method does not take an **Environment** pointer.

Description

If the class that executes this method defines an implementation for the indicated method (because the class either introduces the method, or overrides it), a pointer to code that invokes this implementation is returned. Otherwise, NULL is returned.

Parameters

receiver
A pointer to a class.

method
A method token.

Return Value

A pointer to code that invokes the implementation defined by the receiver for the indicated method. Or, if the receiver does not define an implementation for the method, a null code pointer is returned.

C++ Example

```
#include <somcm.xh>
#include <somcls.xh>
void main()
{
  SOMClassMgr *cmObject = somEnvironmentNew(); // the cm
  SOMClass *cmClass = cmObject->somGetClass(); // the cm class
  SOMClass *cmMeta = cmClass->somGetClass();   // SOMClass
  somTD_SOMObject_somPrintSelf fp = (somTD_SOMObject_somPrintSelf)
     cmMeta->somDefinedMethod(SOMObjectClassData.somPrintSelf);
if (fp)
     fp(cmClass); // output: {The class "SOMClassMgr"}
}
```

Original Class

SOMClass Class

somDescendedFrom Method

Tests whether one class is derived from another. Not generally overridden.

IDL Syntax

boolean somDescendedFrom (in SOMClass *aClassObj*);

Description

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.

Parameters

receiver
A pointer to the class object to be tested.

aClassObj
A pointer to the potential ancestor class.

Return Value

Returns 1 (true) if *receiver* is derived from *aClassObj,* and 0 (false) otherwise.

C Example

```
#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
```

Original Class

SOMClass Class

Related Information

somIsA Method

somIsInstanceOf Method

somFindMethod(Ok) Methods

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

IDL Syntax

boolean somFindMethod (
in somId *methodId*,
out somMethodPtr *m*);
boolean somFindMethodOk (
in somId *methodId*,
out somMethodPtr *m*);

Description

The **somFindMethod** and **somFindMethodOk** methods perform name-lookup method resolution, determine the method procedure appropriate for performing the indicated method on instances of the receiving class, and load *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 methods that return procedure pointers, these methods allow 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.

Parameters

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

methodId
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

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.

Return Value

The **somFindMethod** and **somFindMethodOk** methods return TRUE when the method procedure can be called directly and FALSE when the method procedure is a dispatch function.

Example

Assuming that the *Animal* class introduces the method `setSound`, the type name for the *setSound* method procedure type will be `somTD_Animal_setSound`, as shown below:

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

   myAnimal = AnimalNew();
/* ----------------------------------------
   Note: Next three statements 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!!!
*/
```

Original Class

SOMClass Class

Related Information

somFindSMethod(Ok) Method

somSupportsMethod Method

somClassDispatch Method

somApply Function

somResolve Function

somClassResolve Function

somResolveByName Function

somParentNumResolve Function

SOM_Resolve Macro

SOM_ResolveNoCheck Macro

SOM_ParentNumResolve Macro

somFindSMethod(Ok) Method

Finds the method procedure for a static method. Not generally overridden.

IDL Syntax

somMethodPtr somFindSMethod (in somId *methodId*);
somMethodPtr somFindSMethodOk (in somId *methodId*);

Description

somFindSMethod and **somFindSMethodOk** perform name-lookup resolution in a similar fashion to **somFindMethod** and **somFindMethodOk**, but are restricted to static methods. See the description of **somFindMethod** for a discussion of name-lookup method resolution. Because these methods are restricted to resolving static methods, their interface is slightly different from the **somFindMethod** interfaces; a method procedure pointer is returned when lookup is successful; otherwise NULL is returned.

somFindSMethodOk 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.

Parameters

receiver
A pointer to a class object.

methodId
A somId representing the name of the desired method.

Return Value

The **somFindSMethod** and **somFindSMethodOk** methods return a pointer to the method procedure that supports the specified method for the class.

Example

See **somFindMethod(Ok) Methods** on page 86.

Original Class

SOMClass Class

Related Information

somFindMethod(Ok) Methods

somGetInstancePartSize Method

Returns the total size of the instance data structure introduced by a class. Not generally overridden.

IDL Syntax

long somGetInstancePartSize ( );

Description

somGetInstancePartSize 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.

Parameters

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

Return Value

Returns the size (in bytes) of the instance variables introduced by this class, not its ancestor or descendent classes. If a class introduces no instance variables, 0 is returned.

C Example

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

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

Original Class

SOMClass Class

Related Information

somGetInstanceSize Method

somGetInstanceSize Method

Returns the size of an instance of a class. Not generally overridden.

IDL Syntax

long somGetInstanceSize ( );

Description

The **somGetInstanceSize** method returns the total amount of space needed in an instance of the specified class.

Parameters

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

Return Value

The **somGetInstanceSize** method 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.

C Example

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

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

Original Class

SOMClass Class

Related Information

somGetInstancePartSize Method

somGetInstanceToken Method

Returns a data access token for the instance data introduced by a class. Not generally overridden.

IDL Syntax

somDToken somGetInstanceToken ( );

Description

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. 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.

Parameters

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

Return Value

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

Original Class

SOMClass Class

Related Information

somDataResolve Function

somGetInstancePartSize Method

somGetInstanceSize Method

somGetMemberToken Method

somGetMemberToken Method

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

IDL Syntax

somDToken somGetMemberToken (
long *memberOffset*,
somDToken *instanceToken*);

Description

The **somGetMemberToken** method returns an access token for the data member at offset *memberOffset* within the block of instance data identified by *instanceToken*. 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.

Parameters

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

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

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

Return Value

Returns an access token for the specified data member.

Original Class

SOMClass Class

Related Information

somDataResolve Function

somGetInstancePartSize Method

somGetInstanceSize Method

somGetInstanceToken Method

somGetMethodData 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.

IDL Syntax

boolean somGetMethodData (
in somId *methodId*,
out somMethodData *md*);

Description

The **somGetMethodData** 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.

Parameters

receiver
A pointer to the class that produced the index value.

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

md
A pointer to a *somMethodData* structure.

Return Value

Boolean true if successful; otherwise false.

C++ Example

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

Related Information

somGetMethodData Method

somGetMethodIndex Method

somGetNthMethodInfo Method

somMethodData (somapi.h)

somGetMethodDescriptor Method

Returns the method descriptor for a method. Not generally overridden.

IDL Syntax

somId somGetMethodDescriptor (in somId *methodId*);

Description

The **somGetMethodDescriptor** 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.

Parameters

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

methodId
A **somId** method descriptor.

Return Value

The **somGetMethodDescriptor** method returns a **somId** method descriptor.

Example

```
somId myMethodDescriptor;
myMethodDescriptor = _somGetMethodDescriptor(_Animal,
                         somIdFromString("setSound"));
/* after last use of myMethodDescriptor */
SOMFree (myMethodDescriptor)
```

Original Class

SOMClass Class

Related Information

somAddDynamicMethod Method

somGetMethodData Method

somGetNthMethodData Method

somGetNthMethodInfo Method

somGetMethodIndex Method

Returns a class-specific index for a method. Not generally overridden.

IDL Syntax

long somGetMethodIndex (in somId *methodId*);

Description

The **somGetMethodIndex** 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.

Parameters

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

methodId
A **somId** method ID.

Return Value

The **somGetMethodIndex** method returns a positive long if successful, and a -1 otherwise.

C++ Example

```
#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));
}
```

Original Class

SOMClass Class

Related Information

somGetNthMethodData Method

somGetNthMethodInfo Method

somMethodData (somapi.h)

somGetMethodToken Method

Returns a method access token for a static method. Not generally overridden.

IDL Syntax

somMToken somGetMethodToken (
in somId *methodId*);

Description

somGetMethodToken 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 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.

Parameters

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

methodId
A **somId** identifying a method.

Return Value

The **somGetsMethodToken** method returns a **somMToken** method-access token.

C Example

Assuming that the class `Animal` introduces the method `setSound`,

```
#include <animal.h>
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);
}
```

Original Class

SOMClass Class

Related Information

somClassResolve Function

somParentResolve Function

somResolve Function

somGetMethodData Method

somGetNthMethodInfo Method

somGetName Method

Returns the name of a class. Not generally overridden.

IDL Syntax

string somGetName ( );

Description

somGetName 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 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 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 (for example, the *className***NewClass** procedure, or the _*className* macro). This is because the *className* 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.

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

Parameters

receiver
The class whose name is desired.

Return Value

The **somGetName** method returns a pointer to the name of the class.

Note: The return value of a *remote* object should be freed using **ORBFree()** and the return value of a *local* object should not be freed using **SOMFree()**. While seemingly inconsistent, this was done to maintain backwards compatibility. To simplify coding, invoke **ORBFree()** to free the result whether the object is remote or not (as **ORBFree()** is a no-op in cases where it shouldn't be called).

C++ Example

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

Original Class

SOMClass Class

Related Information

lookup_id Method

somFindClass Method

somSubstituteClass Method

somGetNthMethodData Method

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

IDL Syntax

boolean somGetNthMethodData (
in long *index*,
out somMethodData *md*)

Description

The **somGetNthMethodData** 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.

Parameters

receiver
A pointer to the class that produced the index value.

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

md
A pointer to a somMethodData structure.

Return Value

Boolean true if successful; otherwise, false.

C++ Example

```
#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));
}
```

Related Information

somGetMethodData Method

somGetMethodIndex Method

somGetNthMethodInfo Method

somMethodData (somapi.h)

somGetNthMethodInfo 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.

IDL Syntax

somId somGetNthMethodInfo (
in long *index*,
out somId *descriptor*);

Description

The **somGetNthMethodInfo** 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.

Parameters

receiver
A pointer to the class from which the *index* was obtained using method **somGetMethodIndex**.

index
The *n*th method known to this class, whose method descriptor is desired.

descriptor
A pointer to a **somId** that will be loaded with a **somId** for the descriptor.

Return Value

The **somId** for the indicated method, if a method with the indicated index is known to the receiver; otherwise, NULL.

C++ Example

```
#include <somcls.xh>
main()
{
  somEnvironmentNew();
  somId descriptor, icId = somIdFromString("somNew");
  long ndx = _SOMClass->somGetMethodIndex(icId);
   somId methodId = _SOMClass->somGetNthMethodInfo(ndx,
&descriptor);
  SOM_Test(somCompareIds(icId, methodId));
  SOMFree(icId);
  SOMFree(methodId);
  SOMFree(descriptor);
}
```

Original Class

SOMClass Class

Related Information

somGetMethodIndex Method

somGetNthMethodData Method

Repository (repostry.idl)

somGetNumMethods Method

Returns the number of methods available for a class. Not generally overridden.

IDL Syntax

long somGetNumMethods ( );

Description

The **somGetNumMethods** method returns the number of methods currently supported by the specified class, including inherited methods (both static and dynamic).

The value that the **somGetNumMethods** 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.

Parameters

receiver
A pointer to the class whose instance method count is desired.

Return Value

The **somGetNumMethods** method returns the total number of methods that are currently available for the receiving class.

C Example

```
#include <animal.h>
main()
{
   int numMethods;

    AnimalNewClass(Animal_MajorVersion, Animal_MinorVersion);
   numMethods = _somGetNumMethods(_Animal);
    somPrintf("Number of methods supported by class: %d\n",
              numMethods);
}
```

Original Class

SOMClass Class

Related Information

somGetNumStaticMethods Method

somGetNumStaticMethods Method

Obtains the number of static methods available for a class. Not generally overridden.

IDL Syntax

long somGetNumStaticMethods ( );

Description

The **somGetNumStaticMethods** 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.

Parameters

receiver
A pointer to the class whose static method count is desired.

Return Value

The **somGetNumStaticMethods** method returns the total number of static methods that are available for instances of the receiving class.

C Example

```
#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);
}
```

Original Class

SOMClass Class

Related Information

somGetNumMethods Method

somGetParents Method

Gets a pointer to a class's parent (direct base) classes. Not generally overridden.

IDL Syntax

SOMClassSequence somGetParents ( );

Description

The **somGetParents** method returns a sequence containing pointers to the parents of the receiver.

Parameters

receiver
A pointer to the class whose parent (base) classes are desired.

Return Value

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.

C Example

```
/* Note: Dog is a single-inheritance subclass of Animal. */
#include <dog.h>
main()
{
  Dog myDog;
  SOMClass dogClass;
  SOMClassSequence parents;
  char *parentName;
  int i;

  myDog = DogNew();
  dogClass = _somGetClass(myDog);
  parents = _somGetParents(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
*/
```

Original Class

SOMClass Class

Related Information

somGetClass Method

somGetVersionNumbers Method

Gets the major and minor version numbers of a class's implementation code. Not generally overridden.

IDL Syntax

void somGetVersionNumbers (
out long *majorVersion,*
out long *minorVersion*);

Description

The **somGetVersionNumbers** 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).

Parameters

receiver
A pointer to a class object.

majorVersion
A pointer where the major version number is to be stored.

minorVersion
A pointer where the minor version number is to be stored.

C Example

```
#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);
}
```

Original Class

SOMClass Class

Related Information

somCheckVersion Method

somLookupMethod Method

Performs name-lookup method resolution. Not generally overridden.

IDL Syntax

somMethodPtr somLookupMethod (in somId *methodId*);

Description

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 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.

somLookupMethod 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. Use **somFindMethod** to check the existence of a method.

To use a method procedure pointer as that returned by **somLookupMethod**, it is necessary to typecast the procedure pointer so the compiler can create the procedure call. A programmer making explicit use of this method must 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.

Parameters

receiver
A pointer to the class whose instance method for the indicated method is desired.

methodId
A **somId** of the method whose method-procedure pointer is needed.

Return Value

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.

C++ Example

```
#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);
 }
```

Original Class

SOMClass Class

Related Information

somFindMethod(Ok) Methods

somFindSMethod(Ok) Method

somNew(NoInit) Methods

Creates a new instance of a class.

IDL Syntax

SOMObject somNew ( );
SOMObject somNewNoInit ( );

Description

somNew and **somNewNoInit** 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.

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.

Parameters

receiver
A pointer to the class object that is to create a new instance.

Return Value

A pointer to the newly created **SOMClass** object, or if either of these methods faisl to allocate enough memory for the new object, NULL is returned.

Example

```
#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 */
 }
```

Original Class

SOMClass Class

Related Information

somDefaultInit Method

somRenew(NoInitNoZero) Methods

somRenew(NoInitNoZero) Methods

Creates a new object instance using a passed block of storage.

IDL Syntax

SOMObject somRenew (in somToken *memPtr*);
SOMObject somRenewNoInit (in somToken *memPtr*);
SOMObject somRenewNoInitNoZero (in somToken *memPtr*);
SOMObject somRenewNoZero (in somToken *memPtr*);

Description

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 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 the invoking **somDefaultInit**; **somRenewNoInit** zeros memory, but does not invoke **somDefaultInit**, **somRenewNoInitNoZero** only sets the method table pointer, **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***)**.

Parameters

receiver
A pointer to the class object that is to create the new instance.

memPtr
A pointer to the space to be used to construct a new object.

Return Value

The value of *newObject* is returned, which is now a pointer to a valid, initialized object.

Example

```
#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 &odq.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<5; i++)
       animals[i] =
         _somRenew(animalClass, myAnimalCluster+(i*animalSize));
    /* Uninitialize the animals explicitly: */
    for (i=0; i<5; i++)
       _somUninit(animals[i]);
    /*
     * Finally, the next line frees all 5 animals
     * with one operation.
     */
    SOMFree (myAnimalCluster);
  }
```

Original Class

SOMClass Class

Related Information

somDefaultInit Method

somGetInstanceSize Method

somNew(NoInit) Methods

somSupportsMethod Method

Returns a **boolean** indicating if instances of a class support a static or dynamic method.

IDL Syntax

boolean somSupportsMethod (in somId *methodId*);

Description

The **somSupportsMethod** method determines if instances of the specified class support the specified (static or dynamic) method.

Parameters

receiver
A pointer to the class object to be tested.

methodId
An ID that represents the name of the method.

Return Value

The **somSupportsMethod** method returns 1 (true) if instances of the specified class support the specified method, and 0 (false) otherwise.

Example

```
/* ----------------------------------------------
   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
*/
```

Original Class

SOMClass Class

Related Information

somRespondsTo Method

SOMClassMgr Class

One instance of **SOMClassMgr** is created automatically during SOM initialization. This instance (pointed to by **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. For a sublcass instance to replace the SOM-supplied **SOMClassMgrObject**, use **somMergeInto** to place the current registry information from **SOMClassMgrObject** into your new class-manager object.

Note: **SOMClassMgrObject** is one of the primitive SOM classes. **SOMClassMgrObject** is the instance of **SOMClassMgr** that is generated during SOM initialization that maintains a registry of SOM classes and assists in the dynamic loading and unloading of class libraries.

File Stem

somcm

Base

SOMObject Class

Metaclass

SOMClass Class

Ancestor Classes

SOMObject Class

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 unavailable 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 Function**. 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 new methods introduced by the **SOMClassMgr** Class belong to the following groups.

Group: Basic Functions

somLoadClassFile Method

somLocateClassFile Method

somRegisterClass Method

somUnloadClassFile Method

somUnregisterClass Method

Group: Access

somGetInitFunction Method

somGetRelatedClasses Method

Group: Dynamic

somClassFromId Method

somFindClass Method

somFindClsInFile Method

somMergeInto Method

somSubstituteClass Method

Overridden Methods

somDumpSelfInt Method

somDefaultInit Method

somDestruct Method

somClassFromId Method

Finds a class object, given its **somId**, if it already exists. Does not load the class.

IDL Syntax

SOMClass somClassFromId (in somId *classId*);

Description

Finds a class object, given its **somId**, If it already exists, it does not load the class. Use the **somClassFromId** method instead of **somFindClass** when you do not want the class to automatically load if it does not already exist in the current process.

Parameters

receiver
Usually **SOMClassMgrObject** (or a pointer to an instance of a user-supplied subclass of **SOMClassMgr**).

classId
The **somId** of the class. This can be obtained from the name of the class using the **somIdFromString Function**.

Return Value

Returns a pointer to the class, or NULL if the class object does not yet exist.

C Example

```
#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.
```

Original Class

SOMClassMgr Class

Related Information

somFindClass Method

somFindClsInFile Method

somFindClass Method

Purpose

Finds the class object for a class.

IDL Syntax

SOMClass somFindClass (
in somId *classId*,
in long *majorVersion*,
in long *minorVersion*);

Description

The **somFindClass** method returns the class object for the specified class. This method first uses **somLocateClassFile** 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:

- 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, see "Modifier Statements" on page 133 in *Programmer's Guide for SOM and DSOM*.
- 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 number that is equal to or greater than *minorVersion*.

Parameters

receiver
Usually **SOMClassMgrObject** (or a pointer to an instance of a user-supplied subclass of **SOMClasssMgr**).

classId
The **somId** representing the name of the class.

majorVersion
The class's major version number.

minorVersion
The class's minor version number.

Return Values

A pointer to the requested class object, or NULL if the class could not be found or created.

C Example

```
#include <som.h>
/*
 *  This program creates a class object (from 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);
}
```

This program produces the following output:

```
myClass: Animal
```

Original Class

SOMClassMgr Class

Related Information

somFindClsInFile Method

somLocateClassFile Method

somFindClsInFile Method

Finds the class object for a class, given a filename that can be used for dynamic loading.

IDL Syntax

SOMClass somFindClsInFile (
in somId *classId*,
in long *majorVersion*,
in long *minorVersion*,
in string *file*);

Description

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.

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 number that is equal to or greater than minorVersion.

Parameters

receiver
Usually **SOMClassMgrObject** (or a pointer to an instance of a user-supplied subclass of **SOMClassMgr**).

classId
The **somId** representing the name of the class.

majorVersion
The class's major version number.

minorVersion
The class's minor version number.

file
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`).

Return Value

A pointer to the requested class object, or NULL if the class could not be found or created.

C Example

```
#include <som.h>
/*
 * This program loads a class and creates an instance
 * of ot without requiring the binding (.h) file
 * for the class.
 */
void main()
{
   SOMObject myAnimal;
```

```
   SOMClass animalClass;
   char *animalName = "Animal";
/*
 * Filenames will differ based on platform
 * Set animalfile to "C:\\MYDLLS\\ANIMAL.DLL" for OS/2 and NT.
 * 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.
*/
```

Original Class

SOMClassMgr Class

Related Information

somFindClass Method

somGetInitFunction Method

Obtains the name of the function that initializes the SOM classes in a class library.

IDL Syntax

string somGetInitFunction ( );

Description

The **somGetInitFunction** method supplies the name of the initialization function for 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".

If a 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, 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 **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.

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.

Parameters

receiver
Usually **SOMClassMgrObject** (or a pointer to an instance of a user-supplied subclass of **SOMClassMgr**).

Return Value

Returns a string that names the initialization function of class libraries. By default, this name is the value of the global variable **SOMClassInitFuncName** whose value is **SOMInitModule**.

Original Class

SOMClassMgr Class

Related Information

somFindClass Method

somFindClsInFile Method

SOMInitModule Function

somGetRelatedClasses Method

Returns an array of class objects that were all registered during the dynamic loading of a class.

IDL Syntax

SOMClass * somGetRelatedClasses (in SOMClass *classObj*);

Description

somGetRelatedClasses 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 the **SOMFree Function** 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.

Parameters

receiver
Usually a pointer to **SOMClassMgrObject**, or a pointer to an instance of a user-defined subclass of **SOMClassMgr**.

classObj
A pointer to a **SOMClass** object.

Return Value

The **somGetRelatedClasses** method returns a pointer to an array of pointers to class objects, or NULL, if the specified class was not dynamically loaded.

Example

```
#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);
```

Original Class

SOMClassMgr Class

Related Information

somGetInitFunction Method

somLoadClassFile Method

Dynamically loads a class.

IDL Syntax

```
SOMClass somLoadClassFile (
                    in somId classId,
                    in long majorVersion,
                    in long minorVersion,
                    in string file);
```

Description

The **SOMClassMgr** object uses **somLoadClassFile** 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.

somLoadClassFile 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**.

Parameters

receiver
Usually **SOMClassMgrObject** (or a pointer to an instance of a user-supplied subclass of **SOMClassMgr**).

classId
The **somId** representing the name of the class to load.

majorVersion
The major version number used to check the compatibility of the class's implementation with the caller's expectations.

minorVersion
The minor version number used to check the compatibility of the class's implementation with the caller's expectations.

file
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`).

Return Value

The **somLoadClassFile** method returns a pointer to the class object, or NULL if the class could not be loaded or the class object could not be created.

Original Class

SOMClassMgr Class

Related Information

somFindClass Method

somFindClsInFile Method

somGetInitFunction Method

somUnloadClassFile Method

somLocateClassFile Method

Determines the file that holds a class to be dynamically loaded.

IDL Syntax

```
string  somLocateClassFile (
                      in somId classId,
                      in long majorVersion,
                      in long minorVersion);
```

Description

The **SOMClassMgr** object uses **somLocateClassFile** 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.

Parameters

receiver
Usually **SOMClassMgrObject** (or a pointer to an instance of a user-supplied subclass of **SOMClassMgr**).

classId
The **somId** representing the name of the class to locate.

majorVersion
The major version number used to check the compatibility of the class's implementation with the caller's expectations.

minorVersion
The minor version number used to check the compatibility of the class's implementation with the caller's expectations.

Return Value

The **somLocateClassFile** method returns the name of the file containing the class.

Original Class

SOMClassMgr Class

Related Information

somFindClass Method

somFindClsInFile Method

somGetInitFunction Method

somLoadClassFile Method

somUnloadClassFile Method

somMergeInto Method

Transfers SOM class registry information to another **SOMClassMgr** instance.

IDL Syntax

void somMergeInto (in SOMClassMgr *target*);

Description

somMergeInto 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 new 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 s**omMergeInto** 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 caninvoke **somMergeInto** from an initializer for your new class manager.

Parameters

receiver
: Usually **SOMClassMgrObject** (or a pointer to an instance of a user-supplied subclass of **SOMClassMgr**).

target
: A pointer to another instance of **SOMClassMgr** or one of its subclasses.

C++ Example

```
// === 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;
    NewCM_Init_SOMClassMgr_somDefaultInit(somSelf, ctrl);
    /*
     * local NewCM initialization code added by programmer
     */
    SOMClassMgrObject->somMergeInto(somSelf);
}
// === 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 [   0] 20077A48 00000000 000 001 2008260C SOMObject
 1 [   1] 2007FB38 00000000 000 000 200825EC SOMClassMgr
 1 [   2] 20083B08 00000000 000 004 2008436C NewCM
 1 [   3] 20077BD8 00000000 000 002 2008262C SOMClass
 1 [   4] 20082668 00000000 000 003 2008315C
SOMParentDerivedMetaclass
}
```

Original Class

SOMClassMgr Class

somRegisterClass Method

Adds a class object to the SOM run-time class registry.

IDL Syntax

void somRegisterClass (in SOMClass *classObj*);

Description

s**omRegisterClass** 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.

Parameters

receiver
Usually **SOMClassMgrObject** (or a pointer to an instance of a user-supplied subclass of **SOMClassMgr**).

classObj
A pointer to the class object to add to the SOM class registry.

Original Class

SOMClassMgr Class

Related Information

somUnregisterClass Method

somRegisterClassLibrary Method

Provided for use in SOM Class libraries on platforms that have loader-invoked entry points associated with shared libraries (DLLs).

This function registers a SOM Class Library with the SOM Kernel. The library is identified by its file name and a pointer to its initialization routine. Since this call can occur prior to the invocation of **somEnvironmentNew**, its actions are deferred until the SOM environment has been initialized. At that time, the **SOMClassMgrObject** is informed of all pending library initialization via the **_somRegisterClassLibrary** method. The actual invocation of the library's initialization routine occurs during the execution of the **somFindClass** method (for libraries that are dynamically loaded).

IDL Syntax

void somRegisterClass (in SOMClass *classObj*);

Description

s**omRegisterClass** adds a class object to the SOM runtime class registry maintained by **SOMClassMgrObject**.

All SOM runtime class objects should be registered with the **SOMClassMgrObject**. This is done automatically during the execution of the **somClassReady Method** as class objects are created.

Parameters

receiver
Usually **SOMClassMgrObject** (or a pointer to an instance of a user-supplied subclass of **SOMClassMgr**).

classObj
A pointer to the class object to add to the SOM class registry.

Original Class

SOMClassMgr Class

Related Information

somUnregisterClass Method

somSubstituteClass Method

Causes the **somFindClass**, **somFindClsInFile** and **somClassFromId** methods to substitute one class for another.

IDL Syntax

long somSubstituteClass (
in string *origClassName*,
in string *newClassName*);

Description

somSubstituteClass 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 *newClassName* must be an immediate child of *origClassName*. 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.

Parameters

receiver
Usually **SOMClassMgrObject** or a pointer to an instance of a user-defined subclass of **SOMClassMgr**.

origClassName
A NULL terminated string containing the old class name.

newClassName
A NULL terminated string containing the new class name.

Return Value

The **somSubstituteClass** method returns a value of zero to indicate success; a non-zero value indicates an error was detected.

C Example

```
#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));
}
```

Original Class

SOMClassMgr Class

Related Information

somClassFromId Method

somFindClass Method

somFindClsInFile Method

somMergeInto Method

SOM_SubstituteClass Macro

somUnloadClassFile Method

Unloads a dynamically loaded class and frees the class's object.

IDL Syntax

long somUnloadClassFile (in SOMClass *class*);

Description

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 the **SOMError Function** 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**.

Parameters

receiver
Usually **SOMClassMgrObject** (or a pointer to an instance of a user-su

plied subclass of **SOMClassMgr**).

class
A pointer to the class to be unloaded.

Return Value

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** or NT **FreeLibrary** system call .

Original Class

SOMClassMgr Class

Related Information

somGetRelatedClasses Method

somLoadClassFile Method

somRegisterClass Method

somUnregisterClass Method

somUnregisterClass Method

Removes a class object from the SOM run-time class registry.

IDL Syntax

long somUnregisterClass (in SOMClass *class*);

Description

somUnregisterClass unregisters a SOM class, frees the class object, and unloads the class's dynamically linked library using **somUnloadClassFile**. For every class that was loaded using **somFindClass** or **somFindClsInFile**, you should call **somUnregisterClass** when the class is no longer required.

Parameters

receiver
Usually **SOMClassMgrObject** (or a pointer to an instance of a user-supplied subclass of **SOMClassMgr**).

class
A pointer to the class to be unregistered.

Return Value

The **somUnregisterClass** method returns 0 for a successful completion, or non-zero to denote failure.

Example

```
#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");
}
```

Original Class

SOMClassMgr Class

Related Information

somFindClass Method

somFindClsInFile Method

somLoadClassFile Method

somRegisterClass Method

somUnloadClassFile Method

SOMObject Class

SOMObject is the root class for all SOM classes. 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**.

File Stem

somobj

Metaclass

SOMClass Class

New Methods

The new methods introduced by the **SOMObject** Class belong to the following groups.

Group: Initialization/Termination

somDefaultInit Method

somDefaultAssign Method

somDefaultConstAssign Method

somDefaultConstCopyInit Method

somDefaultCopyInit Method

somDestruct Method

somFree Method

Group: Access

somGetClass Method

somGetClassFromMToken Method

somGetSize Method

Group: Testing

somIsA Method

somIsInstanceOf Method

somRespondsTo Method

Group: Dynamic

somCastObj Method

somClassDispatch Method

somResetObj Method

Group: Development Support

somDumpSelf Method

somDumpSelfInt Method

somPrintSelf Method

Deprecated Methods

Use of the following methods is discouraged:

somDispatchX methods

somInitMethod

somUninit

somCastObj Method

Changes the behavior of an object to that of any ancestor of the true class of the object.

IDL Syntax

boolean somCastObj (in SOMClass *ancestor*);

Description

somCastObj changes the behavior of an object so it will be that of an instance of the indicated ancestor class. The behavior of the object on methods not supported by the ancestor remains unchanged.

This operation changes the class of the object. The name of the new class is derived from the initial name of the object's class and that of the ancestor class.

somCastObj may be used on an object repeatedly with the restriction that the ancestor class whose behavior is chosen is an ancestor of the true (original) class of the object.

Parameters

receiver
A pointer to an object of type **SOMObject**.

ancestor
A pointer to a class that is an ancestor of the actual class of the *receiver*.

Return Value

Returns 1 (TRUE) if the operation is successful and 0 (false) otherwise. The operation fails if ancestor is not actually an ancestor of the class of the object.

Example

```
#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> ...
 *  }
 */
```

Original Class

SOMObject Class

Related Information

somResetObj Method

somDefaultAssign Method

Provides support for an object-assignment operator. May be overridden, but, if appropriate, **somDefaultConstAssign** should be overridden instead.

IDL Syntax

void somDefaultAssign (inout somInitCtrl *ctrl*,
in SOMObject *fromObj*);

Description

In C++, assignment to an object of class X is accomplished by using the assignment operator provided by X. To make assignment available on all SOM objects, **SOMObject** provides **somDefaultAssign** and **somDefaultConstAssign**. 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 **somDefaultConstAssign** are similar to those for overriding **somDefaultInit**. See "Initializing and Uninitializing Objects" on page 195 of *Programmer's Guide for SOM and DSOM* for additional information.

The 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.

Parameters

receiver
A pointer to an object of an arbitrary SOM class, *S*.

ctrl
A pointer to a **somInitCtrl** structure, or NULL.

fromObj
A pointer to an object of class *S* or some class descended from *S*.

Example

```
// 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
}
```

Original Class

SOMObject Class

Related Information

somDefaultAssign Method

somDefaultConstCopyInit Method

somDefaultCopyInit Method

somDefaultInit Method

somDefaultConstAssign Method

Provides support for a “const” object-assignment operator. Designed to be overridden.

IDL Syntax

void somDefaultConstAssign (inout somInitCtrl *ctrl*,
in SOMObject *fromObj*);

Description

In C++, assignments 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 **somDefaultAssign** and **somDefaultConstAssign**. 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 for overriding **somDefaultInit**. See “Initializing and Uninitializing Objects” on page 195 of *Programmer’s Guide for SOM and DSOM* for additional information.

The basic difference between **somDefaultInit** and **somDefaultConstAssign** is that the latter method takes an object (*fromObj*) as an argument that is to be copied.

Parameters

receiver
A pointer to an object of an arbitrary SOM class, `S`.

ctrl
A pointer to a **somInitCtrl** structure, or NULL.

fromObj
A pointer to an object of class *S* or some class descended from *S*.

Example

```
// IDL for a class that overrides somDefaultConstAssign
#include <X.idl>
interface Y : X  {
    implementation  {
        somDefaultConstAssign: override;
    }
}
```

Original Class

SOMObject Class

Related Information

somDefaultAssign Method

somDefaultConstCopyInit Method

somDefaultCopyInit Method

somDefaultInit Method

somDefaultConstCopyInit Method

Provides support for passing objects as call-by-value object parameters. Designed to be overridden.

IDL Syntax

void somDefaultConstCopyInit (inout somInitCtrl *ctrl*, in SOMObject *fromObj*);

Description

somDefaultConstCopyInit 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.

The default behavior of **somDefaultConstCopyInit** is to shallow copy 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**, 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 for overriding **somDefaultInit**. See "Initializing and Uninitializing Objects" on page 195 of *Programmer's Guide for SOM and DSOM* for additional information.

The basic difference between **somDefaultInit** and **somDefaultConstCopyInit** is that the latter method takes an object (*fromObj*) as an argument that is to be copied.

Parameters

receiver
A pointer to an uninitialized object of an arbitrary SOM class, *S*.

ctrl
A pointer to a **somInitCtrl** structure, or NULL.

fromObj
A pointer to an object of class *S* or some class descended from *S*.

Example

```
// IDL for a class that overrides somDefaultConstCopyInit
interface X : SOMObject
{
     implementation {
         somDefaultConstCopyInit: override, init;
     };
};
```

Original Class

SOMObject Class

Related Information

somDefaultAssign Method

somDefaultConstAssign Method

somDefaultCopyInit Method

somDefaultInit Method

somDefaultCopyInit Method

Provides support for call-by-value object parameters. May to be overridden, but, if appropriate, **somDefaultConstCopyInit** should be overridden instead.

IDL Syntax

void somDefaultCopyInit (inout somInitCtrl *ctrl*, in SOMObject *fromObj*);

Description

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.

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 a shallow copy is not appropriate) would it be appropriate to override **somDefaultCopyInit**.

The considerations important to overriding **somDefaultCopyInit** are similar to those described in *Programmer's Guide for SOM and DSOM* for overriding **somDefaultInit**. See **Initializing and Uninitializing Objects** on page 195.

The basic difference between somDefaultInit and somDefaultCopyInit is that the latter method takes an object (fromObj) as an argument that is to be copied.

Parameters

receiver
A pointer to an uninitialized object of an arbitrary SOM class, *S.*

ctrl
A pointer to a **somInitCtrl** structure, or NULL.

fromObj
A pointer to an object of class *S* or some class descended from *S.*

Example

```
// IDL produced for 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
}
```

Original Class

SOMObject Class

Related Information

somDefaultAssign Method

somDefaultConstAssign Method

somDefaultConstCopyInit Method

somDefaultInit Method

somDefaultInit 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.

IDL Syntax

void somDefaultInit (inout somInitCtrl *ctrl*);

Description

Every SOM class is expected to support a set of initializer methods. This set 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. See "Initializing and Uninitializing Objects" on page 195 of *Programmer's Guide for SOM and DSOM* for complete information on introducing new initializers.

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 (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:

- Replace the **somInit** override in the class's .idl file with an override for **somDefaultInit**
- Run the implementation template emitter to produce a stub procedure for **somDefaultInit**
- simply call the class's **somInit** procedure directly 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.

Parameters

receiver
A pointer to an object**.**

ctrl
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.

Example

```
// SOM IDL
#include <Animal.idl>

interface Dog : Animal
{
    implementation {
        releaseorder: ;
            somDefaultInit: override, init;
        };
};
```

Original Class

SOMObject Class

Related Information

somDestruct Method

somDestruct 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.

IDL Syntax

void somDestruct (in octet *dofree*, inout somDestructCtrl *ctrl*);

Description

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 "Initializing and Uninitializing Objects" on page 195 of *Programmer's Guide for SOM and DSOM* 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.

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:

- Replace the **somUninit** override in the class's .idl file with an override for **somDestruct**
- Run the emitter to produce a stub procedure for **somDestruct** in the implementation template file
- Call the class's **somUninit** procedure directly (not using a method invocation) from the **somDestruct** procedure

Parameters

receiver
A pointer to an object.

dofree
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

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.

Example

```
// SOM IDL
#include <Animal.idl>
interface Dog : Animal
{
    implementation {
        releaseorder: ;
            somDestruct: override;
        };
};
```

Original Class

SOMObject Class

Related Information

somDefaultInit Method

somClassDispatch Method

Invokes a method using dispatch method resolution. The **somDispatch** method is designed to be overridden. The **somClassDispatch** method is not generally overridden.

IDL Syntax

boolean somDispatch (
out somToken *retValue*,
in somId *methodId*,
in va_list *args*);
boolean somClassDispatch (
in SOMClass *clsObj*,
out somToken *retValue*,
in somId *methodId*,
in va_list *args*);

Description

Both **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 run-time 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 **somDispatchX** methods. Unlike the **somDispatchX** 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 **somDispatchX** methods now invoke **somDispatch**, so overriding **somDispatch** serves to override the **somDispatchX** methods as well.

Parameters

receiver
A pointer to the object whose class will be used for method resolution by **somDispatch**.

clsObj
A pointer to the class that will be used for method resolution by **somClassDispatch**.

retValue
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 (that is, void), a NULL may be passed as this argument.

methodId
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

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).**

Return Value

Returns a boolean representing if the method was successfully dispatched. **somDispatch** and **somClassDispatch** use **somApply** to invoke the resolved method procedure. **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 and **somApply** might fail. If this happens, then false is returned; otherwise true is returned.

Example

Given class `Key` with an attribute *keyval* of type long and an overridden method for **somPrintSelf** that prints the value of the attribute, 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, 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}
```

Original Class

SOMObject Class

Related Information

somApply Function

somDumpSelf Method

Writes out a detailed description of the receiving object. Intended for use by object clients. Not generally overridden.

IDL Syntax

void somDumpSelf (in long *level*);

Description

somDumpSelf performs some initial setup, and then invokes **somDumpsSelfInt** to write a detailed description of the receiver, including its state.

Parameters

receiver
A pointer to the object to be dumped.

level
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.

Example

See somDumpSelfInt Method on page 152.

Original Class

SOMObject Class

Related Information

somDumpSelfInt Method

somDumpSelfInt Method

Outputs the internal state of an object. Intended to be overridden by class implementors. Not intended to be directly invoked by object clients.

IDL Syntax

void somDumpSelfInt (in long *level*);

Description

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.

The character output routine pointed to by **SOMOutCharRoutine Function** should be used for output. The **somLPrintf Function** is convenient for this, since level is handled appropriately.

Parameters

receiver
A pointer to the object to be dumped.

level
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.

C Example

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");
}
```

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
}
```

Original Class

SOMObject Class

Related Information

somDumpSelf Method

somPrintSelf Method

somFree Method

Releases the storage used by an object and frees the object. Intended for use by object clients. Not generally overridden.

IDL Syntax

void somFree ( );

Description

somFree calls **somDestruct** to allow storage pointed to by the object to be freed. The **somDestruct** method releases the storage containing the receiver object by calling the **somDeallocate Method**. No future references should be made to the receiver.

somFree should not be called on objects created by **somRenew(NoInitNoZero) Methods**, the method is normally used only by code that also created the object.

Note: SOM supplies the **SOMFree** function to free a block of memory. Do not use that function on objects.

Parameters

receiver
A pointer to the object to be freed.

C Example

```
#include <animal.h>
void main()
{
   Animal myAnimal;
   /*
    * Create an object.
    */
   myAnimal = AnimalNew();
   /* ... */
   /* Free it when finished. */
   _somFree(myAnimal);
}
```

Original Class

SOMObject Class

Related Information

somDestruct Method

somNew(NoInit) Methods

SOMFree Function

somGetClass Method

Returns a pointer to an object's class object. Not generally overridden.

IDL Syntax

SOMClass somGetClass ( );

Description

somGetClass obtains a pointer to the receiver's class object. The **somGetClass** method is typically not overridden.

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 (that is, 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.

Parameters

receiver
A pointer to the object whose class is desired.

Return Value

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**.

C Example

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

  myAnimal = AnimalNew ();
  animalClass = _somGetClass (myAnimal);
  SOM_Test(animalClass == _Animal);
}
```

Original Class

SOMObject Class

Related Information

SOM_GetClass Macro

somGetClassFromMToken Method

Provides a public API for determining the introducting class of a method that is indicated by a method token.

IDL Syntax

string somGetClassFromMToken ( );

Description

The **somGetClassFromMToken** method returns a pointer to a zero-terminated string that gives the name of the class of an object.

Parameters

receiver
A pointer to the object whose class name is desired.

Return Value

The **somGetClassFromMToken** method returns a pointer to the name of the class.

Original Class

SOMObject Class

Related Information

somGetName Method

somGetSize Method

Returns the size of an object. Not generally overridden.

IDL Syntax

long somGetSize ( );

Description

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.

Parameters

receiver
A pointer to the object whose size is desired.

Return Value

The **somGetSize** method returns the size, in bytes, of the receiver.

C Example

```
#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
*/
```

Original Class

SOMObject Class

Related Information

somGetInstancePartSize Method

somGetInstanceSize Method

somIsA Method

Tests whether an object is an instance of a given class or a subclasse. Not generally overridden.

IDL Syntax

boolean somIsA (in SOMClass *aClass*)

Description

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 methods supported by *aClass*.

Parameters

receiver
A pointer to the object to be tested.

aClass
A pointer to the class that the object should be tested against.

Return Value

The **somIsA** methods returns TRUE if the receiving object is an instance of the specified class or (unlike **somIsInstanceOf**) of any of its descendant classes, and FALSE otherwise.

Example

In this example, `Dog` is derived from `Animal`.

```
#include <dog.h>
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 */
```

Original Class

SOMObject Class

Related Information

somDescendedFrom Method

somIsInstanceOf Method

somRespondsTo Method

somSupportsMethod Method

somIsInstanceOf Method

Determines whether an object is an instance of a specific class. Not generally overridden.

IDL Syntax

boolean somIsInstanceOf (in SOMClass *aClass*);

Description

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))
```

Parameters

receiver
A pointer to the object to be tested.

aClass
A pointer to the class that the object should be an instance of.

Return Value

The **somIsInstanceOf** method returns 1 (true) if the receiving object is an instance of the specified class, and 0 (false) otherwise.

C Example

In this example, `Dog` is derived from `Animal`.

```
#include <dog.h>
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 */
```

Original Class

SOMObject Class

Related Information

somDescendedFrom Method

somIsA Method

somPrintSelf Method

Outputs a brief description that identifies the receiving object. Designed to be overridden.

IDL Syntax

SOMObject somPrintSelf ( );

Description

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 Function** (or any of the **somPrintf Function**) 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.

Parameters

receiver
A pointer to the object to be described.

Return Value

The **somPrintSelf** method returns a pointer to the receiver object as its result.

C Example

```
#include <animal.h>
main()
{
  Animal myAnimal;
  myAnimal = AnimalNew ();
  /* ... */
  _somPrintSelf (myAnimal);
  _somFree (myAnimal);
}
/*
Output from this program:

{An instance of class Animal at address 0001CEC0}
*/
```

Original Class

SOMObject Class

Related Information

somDumpSelf Method

somDumpSelfInt Method

somResetObj Method

Resets an object's class to its true class after use of the **somCastObj** method.

IDL Syntax

boolean somResetObj ( );

Description

The **somResetObj** method resets an object's class to its true class after use of the **somCastsObj** method.

Parameters

receiver
A pointer to a SOM object.

Return Value

The **somResetObj** method returns 1 (TRUE) always.

Example

```
#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> ...
 *  }
 */
```

Original Class

SOMObject Class

Related Information

somCastObj Method

somRespondsTo Method

Tests whether the receiving object supports a given method. Not generally overridden.

IDL Syntax

boolean somRespondsTo (in somId *methodId*);

Description

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.

Parameters

receiver
A pointer to the object to be tested.

methodId
A **somId** that represents the name of the desired method.

Return Value

The **somRespondsTo** method returns TRUE if the specified method can be invoked on the receiving object, and FALSE otherwise.

C Example

```
/* -----------------------------------------------
   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
*/
```

Original Class

SOMObject Class

Related Information

somSupportsMethod Method

Chapter 2. DSOM Framework

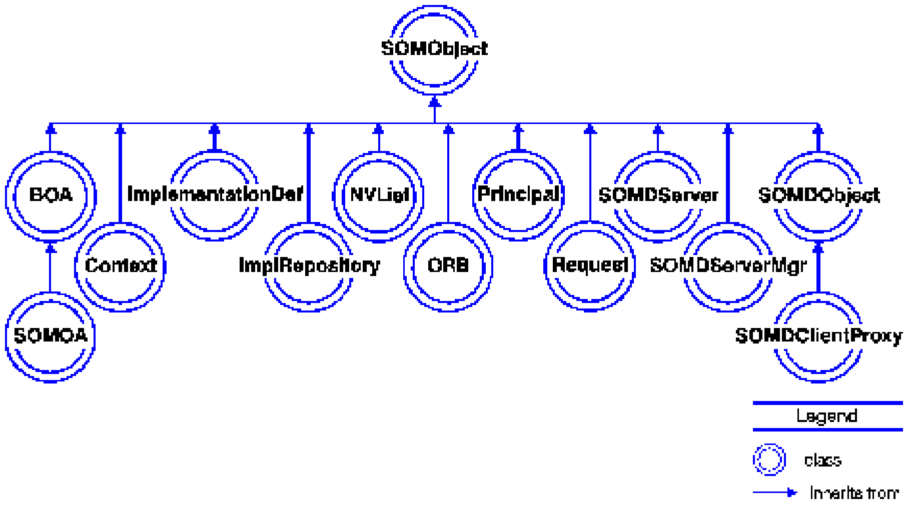

Figure 2. DSOM Framework Organization.

Notes About DSOM and CORBA

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 OMG.

One of the primary contributions of CORBA 1.1 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 1.1 specification.

In addition to the published CORBA 1.1 interfaces, it was necessary for DSOM to introduce several of its own interfaces, in those areas where:

- CORBA 1.1 did not specify the full interface (for example, **ImplementationDef Class**, **Principal Class**)
- CORBA 1.1 did not address the function specified by the interface.
- 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.

Method Naming Conventions

The SOM Toolkit frameworks, including DSOM, and CORBA 1.1 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, **somdDispatchMethod**). Methods introduced by CORBA 1.1 have no prefixes, are all lower 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 1.1-defined class also use the CORBA 1.1 naming style. The SOM Toolkit convention for method naming is used for non-CORBA 1.1 classes which are introduced by DSOM.

get_next_response Function

Returns the next **Request Class** object to complete, after starting multiple requests in parallel.

C Syntax

ORBStatus get_next_response (
Environment* ***ev,***
Flags ***response_flags,***
Request *****req* **);**

Description

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.

This function is described in Deferred Synchronous Routines, of the CORBA 1.1 specification.

Parameters

ev
A pointer to the **Environment** structure for the caller.

response_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
A pointer to a **Request** object variable. The address of the next **Request** object which completes is returned in the **Request** variable.

Return Value

The **get_next_response** function may return a non-zero **ORBStatus** value, which indicates a DSOM error code. (DSOM error codes are listed in “Error Codes” on page 399 of *Programmer’s Guide for SOM and DSOM.*)

Example

See **send_multiple_requests Function** on page 170.

Related Information

send_multiple_requests Function

get_response Method

invoke Method

send Method

ORBfree Function

Frees memory allocated in a DSOM client by DSOM for **out** arguments and certain return values from remote method calls. Also frees memory returned from methods invoked on the **ORB** object.

C Syntax

void ORBfree (void * *ptr*);

Description

The purpose of **ORBfree** is to recursively free all of the storage associated with out parameters or certain return values from remote method calls. Such storage is allocated by the DSOM run time in the client's address space on behalf of the remote object. Storage so allocated must be given special treatment by the user, specifically pointers within it may not be modified nor freed using **SOMFree**. They must be freed using **ORBfree**.

By default, for remote method calls all **out** parameters and the following types of return values must be freed with **ORBfree**:

returned strings

returned pointers (that is, all "*" types)

returned arrays

returned sequences

If there are object references or **TypeCode** references within the storage, **ORBfree** will appropriately release the references. If a given **out** parameter requires no allocation by DSOM (as is the case for an **out long** parameter, for example), it is unnecessary but has no effect to call **ORBfree** on it.

ORBfree does not apply to object-owned parameters, **in** or **inout** parameters, or return values of types not listed above. These must always be freed by using **SOMFree** on each of the contained pointers. **SOMD_FreeType** can help with this. **ORBfree** never applies to parameters of local method calls nor to memory allocated by the application itself.

The need for **ORBfree** can be disabled by calling the function **SOMD_NoORBfree** prior to making the remote method call. **SOMD_NoORBfree** allows an application to achieve local/remote transparency. If **SOMD_NoORBfree** has been called, an application can free results of both local and remote method calls in the same way (using **SOMFree** or **SOMD_FreeType**).

The **ORBfree** function is used to free the storage returned from certain **ORB** methods, including **object_to_string** and **list_initial_services**.

Parameters

ptr

A pointer to memory that has been allocated by DSOM for an **out** argument or a method return value. Since **ORBfree** applies to a whole parameter, the *ptr* argument is the top-level pointer used to return the parameter (as required by CORBA 1.1, section 5.16, pg 96). Specifically, for **out** parameters the *ptr* argument to **ORBfree** is the pointer used to return the **out** parameter. For return values (except **sequences**), the *ptr* argument is the returned **pointer** or object references. For returned **sequences**, the argument to **ORBfree** should be the *_buffer* field of the **sequence**.

Return Value

None. There is currently no way to determine whether a given pointer corresponds to a parameter that **ORBfree** should free (whether the call was successful) because the signature of **ORBfree** is specified by CORBA 1.1.

Example

```
#include <somd.h>
#include <myobject.h>  /* provided by user */

MyObject obj;
Environment ev;
string str1, str2;
MyStruct m;
/* Assume the following appears in the IDL for the
 * MyObject interface:
 *
 *      struct MyStruct {
 *          long i;
 *          long j;
 *      };
 *
 *      string myMethod(out string s, out MyStruct m);
 */
SOM_InitEnvironment (&ev);
/* Assume obj is a proxy for a remote MyObject */
str1 = _myMethod(obj, &ev, &str2, &m);
...
/* Free storage */
ORBfree(str1);  /* argument is the returned pointer */
ORBfree(&str2); /* argument is ptr used to return out */
ORBfree(&m);    /* unnecessary, but has no effect */
```

Related Information

SOMD_FreeType Function

SOMD_NoORBfree Function

SOMD_QueryORBfree Function

SOMD_YesORBfree Function

SOMFree Function

list_initial_services Method

object_to_string Method

This function is described in "Argument Passing Considerations" and section 5.17, “Return Result Passing Considerations,” of the CORBA 1.1 specification.

send_multiple_requests Function

Initiates multiple **Request Class** in parallel.

C Syntax

ORBStatus send_multiple_requests (
Request ***reqs[ ],***
Environment* ***ev,***
long ***count,***
Flags ***invoke_flags*** **);**

Description

The **send_multiple_requests** function initiates multiple **Request** in parallel. (The actual degree of parallelism is system dependent.) Each **Request** object is created using the **create_request Method** method, defined on **SOMDClientProxy Class**. Like the **send** method, this function returns to the caller immediately without waiting for the **Request** to finish. The caller waits for the request responses using the **get_next_response** function.

Parameters

reqs
The address of an array of **Request** objects which are to be initiated in parallel.

ev
A pointer to the **Environment** structure for the caller.

count
The number of **Request** objects in *reqs.*

invoke_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.

Return Value

The **send_multiple_requests** function may return a non-zero **ORBStatus** value, which indicates a DSOM error code. (DSOM error codes are listed in “Error Codes” on page 399 of *Programmer’s Guide for SOM and DSOM.*)

Example

```
#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);
}
```

Related Information

get_next_response Function

get_response Method

invoke Method

send Method

This function is described in section 6.3, “Deferred Synchronous Routines”, of the CORBA 1.1 specification.

somdCreate Function

Requests creation of an object of a specified class, by any available factory.

C Syntax

SOMObject * somdCreate (
**Environment ****ev*,
Identifier ***className***
boolean ***init*****);**

Description

The **somdCreate** function is provided as a convenience function for object creation; it searches for any factory that knows how to create objects of the requested class and asks the factory to create such an object. The **somdCreate** function calls **find_any** requesting that property *class* be set to the input *className*. The function call also specifies the method that should be used for object creation: **somNew** or **somNewNoInit.**

Parameters

ev
A pointer to the **Environment** structure used to return errors.

className
The class name of the object to be created. The *className* parameter must match the class name as specified when the class was associated with some server (for example, via the **regimpl** tool).

init
A boolean value that specifies how to create the object; TRUE means to call **somNew**, or FALSE means to call **somNewNoInit.**

Return Value

The **somdCreate** function returns a pointer to the newly created object of the specified class.

Example

```
#include <somd.h>
#include <car.h>
Environment ev;
SOM_InitEnvironment (&ev);
/* create an instance of class "Car" using somNew */
car = somdCreate(&ev, "Car", TRUE);
```

Related Information

somNew(NoInit) Methods

find_any Method

somdCreateDynProxyClass Function

Creates a DSOM proxy class.

C Syntax

SOMClass * somdCreateDynProxyClass (string targetClassName*)*

Description

This method creates a DSOM proxy class for the specified target class which is useful when implementing a factory that controls memory allocation for both local objects and proxies for remote objects. See "Designing Local/Remote Transparent Programs" on page 271 for more information.

For more details about constructing DSOM proxy classes, see "Object References and Proxy Objects" on page 329.

Parameters

ev
A pointer to the **Environment** structure for the method caller.

targetClassName
The name of a SOM class from which the proxy class will be derived. The proxy class inherits both interface and implementation from the SOMDClientProxy class, and inherits interface only from the class specified by *targetClassName*.

If the IDL for the specified *targetClassName* designates an application-specific baseproxyclass, that proxy base class is used as the parent class instead of SOMDClientProxy.

Return Value

Returns the constructed proxy class object named with the *targetClassName* concatenated with the literal string Proxy. The returned class object name appears as *targetClassName*__Proxy: note the double underscore between *targetClassName* and Proxy. If a class with the same name already exists, that constructed proxy class object is returned.

If the proxy class cannot be constructed, a system exception is returned in the **Environment**. For example, if the class for the specified *targetClassName* cannot be created because its DLL cannot be loaded by **somFindClass**, a system exception is returned.

somdDaemonReady Function

Checks whether **somdd**, the DSOM daemon, is running.

C Syntax

boolean somDaemonReady (
**Environment ****ev*,
long ***timeout*****);**

Description

The **somdDaemonReady** function checks whether the DSOM daemon, **somdd**, is running and ready to accept requests. The function checks for a **somdd** executable active on the same system as the query.

If the daemon is not running, **somdDaemonReady** checks the status of the daemon for *timeout* seconds. A message is displayed periodically while waiting for the daemon to become active.

Parameters

ev
A pointer to the **Environment** structure used to return errors.

timeout
The number of seconds to wait for the caller**.**

Return Value

The function returns TRUE if the DSOM daemon is running and ready to accept requests. If the daemon is not running FALSE is returned and an exception is returned in *ev*.

Example

```
#include <somd.h>
Environment ev;
SOM_InitEnvironment (&ev);
if (somdDaemonReady (&ev, 0) {
/* somdd is up and running */
}
```

somdExceptionFree Function

Frees the memory held by the exception structure within an **Environment** structure that was returned from a remote method invocation.

C Syntax

void somdExceptionFree (Environment ****ev*);**

Description

The **somdExceptionFree** function frees the memory held by the exception structure within an **Environment** structure that was returned from a remote method invocation.

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. These two functions are equivalent. The **exception_free** function name is **#define**d in the **som.h** or **som.xh** file to provide strict CORBA 1.1 compliance of function names.

The **somdExceptionFree** function does a recursive (deep) free of the exception's parameters (similar to **ORBfree Function** for method return results), unless **SOMD_NoORBfree Function** was invoked prior to the remote call or the exception was returned from a local call. The exception name is also freed, and the **Environment** structure's fields are reset to indicate no exception.

This function should not be invoked on irOpenError exceptions returned from the Interface Repository framework because the Interface Repository framework allocates all exception parameters from a single block of memory; therefore, these exception parameters should not be recursively freed, but should instead be freed using **somExceptiomFree** or a single call to **SOMFree**.

If **SOMD_NoORBfree** has been invoked, or if the exception was returned from a local (non-remote) method call, then **somdExceptionFree** performs only a shallow free. If the exception parameters contain embedded memory blocks, then the application should explicitly free the additional memory using calls to **SOMFree Function** for each memory block contained therein. If the exception parameters consist of a single block of memory, then the exception will be completely freed by **somdExceptionFree** or **exception_free**.

A similar function, **somExceptionFree**, is available for SOM applications (those that do not include DSOM header files). For SOM programmers, **exception_free** is **#define**d to **somExceptionFree** rather than **somdExceptionFree**. The **somExceptionFree** function (like **somdExceptionFree** when **SOMD_NoORBfree** has been invoked, and like **somdExceptionFree** when invoked on an exception returned from a local method call) does not do a deep free (under any circumstances), and thus does not completely free exceptions whose parameters contain multiple blocks of memory.

Parameters

ev

The **Environment** structure whose exception information is to be freed. If ev is NULL, then the global Environment's exception structure is freed. It is an error to invoke this function on an **Environment** structure that does not contain an exception.

Example

```
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 */
    }
```

Related Information

somdExceptionFree Function

somExceptionId Function

somExceptionValue Function

somSetException Function

Environment (som.h)

SOMD_FlushInterfaceCache Function

Removes entries from DSOM's internal cache of Interface Repository entries.

C Syntax

SOMEXTERN void SOMLINK SOMD_FlushInterfaceCache (
Environment *_ev_,
string _name_);

Description

DSOM maintains an internal cache of the Interface Repository entries that it references, in addition to the caching done by the Interface Repository framework itself. The function **SOMD_FlushInterfaceCache** allows users to purge this cache, either entirely or for selected items. This function complements the **Repository::release_cache** method that purges the cache maintained by the Interface Repository framework. The **SOMD_FlushInterfaceCache** function invokes **Repository::release_cache** before returning, to ensure that updated information in the IR will be used when DSOM subsequently accesses the IR.

Parameters

ev
A pointer to an Environment structure in which exception information is returned.

name
The name of the Interface Repository entry (interface name, method name, or attribute name) to be removed from DSOM's internal cache. If this name is not specified, then all entries in DSOM's internal cache are purged.

Example

```
SOMD_FlushInterfaceCache(ev, ”changedMethod”);
```

Related Information

release_cache Method

SOMD_FreeType Function

Deep frees memory for an instance of a specified type.

C Syntax

void SOMD_FreeType (
Environment * ***ev*,**
void * ***valptr*,**
TypeCode *type*);

Description

SOMD_FreeType takes a pointer to a value, together with the **TypeCode** for its type, and frees all the freeable storage occupied by the value. **SOMD_FreeType** does not free the storage pointed to by *valptr* itself, since that may not be freeable (see the Example that follows).

SOMD_FreeType executes by "walking" the value and calling **SOMFree** on each contained block of memory (as well as calling the appropriate **release** method on each contained object or pseudo-object reference). Thus, **SOMD_FreeType** is applicable to any storage that was allocated using the standard means, but not to storage that must be freed with **ORBfree**.

SOMD_FreeType can return an exception in *ev*, usually because the provided **TypeCode** does not match the value. If **SOMD_FreeType** returns an exception, the storage for the value is left in an undefined state.

Parameters

ev
A pointer to the **Environment** structure for the caller, where **SOMD_FreeType** may return an exception.

valptr
A pointer to the value whose storage is to be freed.

type
The **TypeCode** for the type of the value to be freed. TypeCodes can be obtained via the Interface Repository framework.

Example

```
#include <somd.h>
#include <myobject.h>  /* provided by user */
MyObject obj;
Environment ev;
any a;
MyRec m;
/* Assume the following appears in the IDL for the
 * MyObject interface:
 *
 *      struct MyRec {
 *          string name;
 *          string address;
 *      };
 *
 *      any myMethod(out MyRec r);
 */
SOM_InitEnvironment (&ev);
/* Assume that the IDL was compiled with -mtcconsts,
 * so that TC_MyObject_MyRec is the TypeCode for MyRec.
```

```
 *
 * Assume that MyObject is local, or that SOMD_NoORBfree
 * is in effect (so that memory allocation is standard).
 */
a = _myMethod(obj, &ev, &m);
...
/* Free strings in m */
SOMD_FreeType(&ev, &m, TC_MyObject_MyRec);
/* Free the storage in the value of the any a */
SOMD_FreeType(&ev, a._value, a._type);

/* Free the storage a._value points to (not freed above) */
SOMFree(a._value);

/* Free the _type field of a */
TypeCode_free(a._type, ev);
```

Related Information

ORBfree Function

SOMFree Function

SOMD_Init Function

Initializes DSOM in the calling process.

Note: This function employs *reference counting* which means that calling **SOMD_Init** multiple times will increment the count.

C Syntax

void SOMD_Init (Environment* *ev*);

Description

Initializes DSOM in the calling process. This function should be called before any other DSOM functions or methods. This function should only be invoked:

- at the beginning of a DSOM program (client or server), to initialize the program
- after **SOMD_Uninit Function** 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 variable **SOMD_ORBObject** is initialized with a pointer to the (single) instance of the ORB object. Also the global variable **SOMD_ImplRepObject** is initialized with a pointer to the (single) instance of the **ImplRepository Class**.

Parameters

ev
A pointer to the **Environment** structure for the caller.

Example

```
#include <somd.h>
Environment ev;
/* initialize Environment */
SOM_InitEnvironment(&ev);
/* initialize DSOM runtime */
SOMD_Init(&ev);
...
/* Free DSOM resources */
SOMD_Uninit(&ev);
```

Related Information

See "Distributed SOM" on page 229 in *Programmer's Guide for SOM and DSOM*.

SOMD_NoORBfree Function

Lets client programs free memory returned from method calls without knowing which calls are local versus remote. Specifies to DSOM that it should not use special allocation techniques to allocate storage for caller-owned **out** parameters and certain return values of remote method calls. Specifies to DSOM that the client program will use **SOMFree** rather than **ORBfree** to free memory allocated by DSOM. Also specifies that **exception_free** (**somdExceptionFree Function**) will not deep-free exceptions.

Note: **exception_free** and **somdExceptionFree** are equivalent. **exception_free** name is #defined in the **som.h** or **som.xh** file to provide strict CORBA 1.1 compliance of function names.

C Syntax

void SOMD_NoORBfree ();

Description

SOMD_NoORBfree allows client programs to free memory returned from a method call whether the memory is remote versus local. **SOMD_NoORBfree** specifies to DSOM that it should not use special allocation techniques to allocate storage for caller-owned **out** parameters and certain return values for remote method calls, and that **ORBfree** will not be called by the application. Issuing **SOMD_NoORBfree** is an indication that the client program wants to treat storage for these parameters uniformly with storage allocated by the program and intends to free it with **SOMFree** or **SOMD_FreeType**. Consequently, calling **ORBfree** has no effect on memory returned when **SOMD_NoORBfree** is in effect.

Similarly, **SOMD_NoORBfree** indicates that DSOM should not use special allocation for exceptions, and that the client will take responsibility for freeing any exception params. When **SOMD_NoORBfree** is in effect, **somdExceptionFree** has the same behavior as **somdExceptionFree Function**.

By default, a DSOM client program must use **SOMFree** to free memory returned by local method calls and **ORBfree** to free certain memory returned from *remote* method calls. Also by default, **somdExceptionFree** will shallow free exceptions returned from local calls, and will deep free exceptions from remote calls. By contrast, **SOMD_NoORBfree** allows programmers to treat memory and exceptions returned from remote method calls uniformly with locally allocated memory (that is, to free memory with **SOMFree** or **SOMD_FreeType**) .

SOMD_NoORBfree can be called any time after **SOMD_Init Function**. However, it will affect the process global state; thus you must exercise caution when using it in multi-threaded clients. The simplest usage is to call **SOMD_NoORBfree** after **SOMD_Init**.

Example

```
SOMD_Init(ev);
SOMD_NoORBfree();
/* rest of client program */
```

Related Information

ORBfree Function

SOMD_FreeType Function

SOMD_QueryORBfree Function

SOMD_YesORBfree Function

SOMFree Function

SOMD_QueryORBfree Function

Queries to determine whether **SOMD_NoORBfree** is in effect.

C Syntax

boolean SOMD_QueryORBfree ();

Description

SOMD_QueryORBfree queries the process global state. Programmers must exercise caution when using it in multi-threaded clients. In a multi-threaded client, this function (like the **SOMD_NoORBfree** and **SOMD_YesORBfree** functions) must be treated as a shared resource and protected by a mutex semaphore.

Return Value

SOMD_QueryORBfree returns TRUE if DSOM is in its default state for allocation of certain parameters of remote calls, or FALSE if the **SOMD_NoORBfree** function is in effect.

Example

```
/* Assume that this program is single-threaded */

SOMD_Init(ev);

/* next call returns TRUE */
b = SOMD_QueryORBfree();
SOMD_NoORBfree();

/* next call returns FALSE */
b = SOMD_QueryORBfree();
SOMD_YesORBfree();

/* next call returns TRUE */
b = SOMD_QueryORBfree();
```

Related Information

ORBfree Function

SOMD_FreeType Function

SOMD_NoORBfree Function

SOMD_YesORBfree Function

SOMFree Function

SOMD_Uninit Function

Frees system resources allocated for use by DSOM.

Note: This function employs *reference counting* which means that **SOMD_Uninit** does not free system resources until the count reaches zero.

C Syntax

void SOMD_Uninit (Environment* *ev*);

Description

Frees system resources (shared memory segments, semaphores, and so forth) 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 Function**. (**SOMD_Uninit** would then need to be called again before program termination to uninitialize the program.)

Parameters

ev
A pointer to the **Environment** structure for the method caller.

Example

```
#include <somd.h>

Environment ev;

/* initialize Environment */
SOM_InitEnvironment(&ev);

/* initialize DSOM runtime */
SOMD_Init(&ev);
...
/* Free DSOM resources */
SOMD_Uninit(&ev);
```

Related Information

See **"Distributed SOM" on page 229** in *Programmer's Guide for SOM and DSOM.*

SOMD_YesORBfree Function

Undoes the effect of **SOMD_NoORBfree**.

C Syntax

void SOMD_YesORBfree ();

Description

The **SOMD_YesORBfree** function negates the effect of the **SOMD_NoORBfree** function, and thus returns DSOM to its default behavior for allocating certain parameters on remote method calls. (For details, see the descriptions of the **ORBfree** and **SOMD_NoORBfree** functions.)

Calling **SOMD_YesORBfree** without having first called **SOMD_NoORBfree** has no effect.

SOMD_YesORBfree affects the process global state; thus programmers must exercise caution when using it in multi-threaded clients. Essentially, in a multi-threaded client **SOMD_YesORBfree** (as well as **SOMD_NoORBfree** and **SOMD_QueryORBfree**) must be treated as a shared resource and protected by a mutex semaphore.

Example

```
#include <somd.h>
#include <myobject.h>  /* provided by user */

MyObject obj;
Environment ev;
string str1, str2;
/* Assume the following appears in the IDL for the
 * MyObject interface:
 *      void myMethod(out string s);
 * Assume also that obj is a proxy for a remote MyObject,
 * and that this program is single-threaded.
 */
SOM_InitEnvironment (&ev);
SOMD_Init(&ev);
SOMD_NoORBfree();

_myMethod(obj, &ev, &str1);
SOMD_YesORBfree();

_myMethod(obj, &ev, &str2);

/* Free storage */
SOMFree(str1);
ORBfree(&str2);
```

Related Information

ORBfree Function

SOMD_FreeType Function

SOMD_NoORBfree Function

SOMD_QueryORBfree Function

SOMFree Function

Context_delete Macro

Deletes a **Context Class** object.

Syntax

ORBStatus Context_delete (
Context *ctxobj*,
Environment ****ev*,
Flags *del_flag*);

Description

The **Context_delete** macro deletes the specified **Context** object. This macro maps to the destroy method of the **Context** class.

Parameters

ctxobj
A pointer to the **Context** object to be deleted.

ev
A pointer to the **Environment** structure for the caller.

del_flag
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.

Expansion

Context_destroy *(ctxobj, ev, del_flag)*

Example

```
#include <somd.h>

Environment ev;
Context cxt, newcxt;
long rc;
SOM_InitEnvironment (&ev);
...
/* 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);
```

Related Information

destroy Method

Request_delete Macro

Deletes the memory allocated by the ORB for a **Request Class** object.

Syntax

ORBStatus Request_delete (
Request *reqobj*,
Environment **ev*);

Description

The **Request_delete** macro deletes the specified **Request** object and all associated memory. This macro maps to the destroy method of the **Request** class.

Parameters

reqobj
A pointer to the **Request** object to be deleted.

ev
A pointer to the **Environment** structure for the caller.

Expansion

Request_destroy *(reqobj, ev)*

Example

```
#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 this 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;
SOM_InitEnvironment (&ev);
/* 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);
```

Related Information

destroy Method

BOA Class

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. The methods of **BOA** are intended to be called from a user-written server program (such as, **deactivate_impl** and **impl_is_ready**), from a user-written subclass of **SOMDServer** (such as, **create**, **dispose**, and **get_id**), or from an application object running within a server (for example, **set_exception** and **get_principal**).

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 1.1 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.

File Stem

boa

Base

SOMObject Class

Metaclass

SOMMSingleInstance Metaclass

Ancestor Classes

SOMObject Class

Subclasses

SOMOA Class

New Methods

change_implementation Method

create Method

deactivate_impl Method

deactivate_obj Method

dispose Method

get_id Method

get_principal Method

impl_is_ready Method

obj_is_ready Method

set_exception Method

change_implementation Method

Note: This method is not implemented.

Changes the implementation definition associated with the referenced object.

IDL Syntax

```
void change_implementation (
        in SOMDObject obj,
        in ImplementationDef impl);
```

Description

The **change_implementation** method is defined by the CORBA 1.1 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 Class** identifies the server which implements an object. In these terms, changing an object's implementation 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 Class.** To change the implementation of an object's class, replace the corresponding class library with a new (upward-compatible) one.

Parameters

receiver
A pointer to a BOA (SOMOA) object for the server.

ev
A pointer to the **Environment** structure for the method caller.

obj
A pointer to the **SOMDObject Class** object which refers to the application object whose implementation is to be changed.

impl
A pointer to the **ImplementationDef** object representing the new implementation of the application object.

Return Value

The **SOMOA Class** implementation always returns a NO_IMPLEMENT exception, with a minor code of SOMDERROR_NotImplemented.

Original Class

BOA Class

create Method

Creates a “reference” for a local application object that can be exported to remote clients.

IDL Syntax

```
typedef sequence<octet,1024> ReferenceData;     // in somdtype.idl
SOMDObject  create (
                        in ReferenceData id,
                        in InterfaceDef intf,
                        in ImplementationDef impl);
```

Description

The **create** method creates a **SOMDObject Class** 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: think of it as an “id”, “link” or “handle.” Object references are important in DSOM in that their values can be externalized (that is, 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.

This method is intended to be called primarily from user-written subclasses of **SOMDServer Class**, within overrides of the **somdRefFromSOMObj** method.

To create an object reference, the caller specifies the **ImplementationDef Class** of the calling process, the **InterfaceDef Class** 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.

Ownership of the returned **SOMDObject** is transferred to the caller.

In DSOM 2.x, the **create** and **create_constant** methods served different purposes. In the current release, however, the methods are equivalent, except for servers whose ImplementationDef object (from the Implementation Repository) specifies an object reference table filename. (To retain the DSOM 2.x semantics for the create method, simply specify a reference table filename for the server using the **regimpl -f** *filename* argument. Otherwise, SOMOA's implementation of the create method will simply invoke **SOMOA::create_constant**.) In addition, the DSOM 2.x methods **SOMOA::change_id** and **SOMOA::create_constant**, and the use of the Object Reference Table, have been deprecated. (They are supported in this release, but may be eliminated in a future release.)

Eventually, the non-CORBA-compliant **SOMOA::create_constant** will be eliminated in favor of the CORBA 1.1-compliant **create** method. Servers that need persistent storage of object-reference data, such as that previously provided by the Object Reference Table, should implement it in an application-specific subclass of **SOMDServer**, or use the **somOS::Server** class for this purpose.

Parameters

receiver
: A pointer to a BOA (SOMOA) object for the server.

ev
: A pointer to the **Environment** structure for the method caller.

id
A pointer to the **ReferenceData** structure containing application-specific information describing the target object.

intf
A pointer to the **InterfaceDef** object that describes the interface of the target object, obtained from the Interface Repository.

impl
A pointer to the **ImplementationDef** object that describes the application (server) process that implements the target object (usually the global variable SOMD_ImplDefObject).

Return Value

The **create** method returns a pointer to a **SOMDObject** which refers to a local application object.

Example

```
/* Inside an implementation of somdRefFromSOMObj
 * in a user-defined subclass of SOMDServer:     */

#include <repostry.h>
#include <intfacdf.h>

Environment ev;
ReferenceData id;
InterfaceDef intfdef;
SOMDObject objref;
string fname; /* a file name to be saved with reference */

...
SOM_InitEnvironment (&ev);
SOMD_Init(&ev);
/* 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);
```

Original Class

BOA Class

Related Information

create_constant Method

create_SOM_ref Method

dispose Method

get_id Method

somdRefFromSOMObj Method

deactivate_impl Method

Indicates that a server implementation is no longer ready to process requests.

IDL Syntax

void deactivate_impl (in ImplementationDef *impl*);

Description

The **deactivate_impl** method indicates that the implementation is no longer ready to process requests. This method should be invoked by every server process prior to termination (whether normal or abnormal) once that server has called **impl_is_ready**. If **impl_is_ready** has not yet been invoked when the server terminates, then the server should invoke **activate_impl_failed** instead.

Parameters

receiver
A pointer to a BOA (SOMOA) object for the server.

ev
A pointer to the **Environment** structure for the method caller.

impl
A pointer to the **ImplementationDef** object representing the implementation to be deactivated usually the **SOMD_ImplDefObject** global variable.

Example

```
#include <somd.h>

Environment ev;
ORBStatus rc;

SOM_InitEnvironment (&ev);
SOMD_Init(&ev);
/* 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);
```

Original Class

BOA Class

Related Information

activate_impl_failed Method

execute_request_loop Method

execute_next_request Method

impl_is_ready Method

deactivate_obj Method

Indicates that an object server is no longer ready to process requests.

Note: Not implemented.

IDL Syntax

void deactivate_obj (in SOMDObject *obj*);

Description

The **deactivate_obj** method is defined by the CORBA 1.1 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 **deactivate_obj** method is meant to be used by unshared servers, to indicate that the object (that is, 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.

Parameters

receiver
A pointer to a BOA (SOMOA) object for the server.

ev
A pointer to the **Environment** structure for the method caller.

obj
A pointer to a **SOMDObject** which identifies the object (server) to be deactivated.

Original Class

BOA Class

Related Information

deactivate_impl Method

impl_is_ready Method

obj_is_ready Method

dispose Method

Destroys an object reference.

IDL Syntax

void dispose (in SOMDObject *obj*);

Description

The **dispose** method disposes of an object reference. This removes data for the object reference from the Object Reference Table (if necessary) and releases the object reference.

Parameters

receiver
A pointer to a BOA object for the server.

ev
A pointer to the **Environment** structure for the method caller.

obj
A pointer to the object reference to be destroyed.

Example

```
#include <somd.h>
#include <repostry.h>
#include <intfacdf.h>

Environment ev;
SOMDObject objref;
ReferenceData id;
InterfaceDef intfdef;

SOM_InitEnvironment (&ev);
SOMD_Init(&ev);
.../*server initialization code*/
objref =
 _create(SOMD_SOMOAObject, &ev, id, intfdef, SOMD_ImplDefObject);
...
_dispose(SOMD_SOMOAObject, &ev, objref);
```

Original Class

BOA Class

Related Information

create Method

create_constant Method

create_SOM_ref Method

get_id Method

get_id Method

Returns reference data associated with the referenced object.

IDL Syntax

ReferenceData get_id (
in SOMDObject *obj*);

Description

The **get_id** method returns the reference data associated with the referenced object (as specified to the call to **create** or **create_constant** method when the object reference was created). This is useful in subclasses of **SOMDServer Class**, in overrides of the **somdSOMObjFromRef** method.

This method should not be invoked on **OBJECT_NIL**, on a proxy object, or on an object for which **SOMOA::is_SOM_ref** returns TRUE. In other words, a subclass of **SOMDServer** should only invoke **get_id** on a **SOMDObject** that it initially created in its own override of the **somdRefFromSOMObj** method.

Parameters

receiver
A pointer to a BOA (SOMOA) object for the server.

ev
A pointer to the **Environment** structure for the method caller.

obj
A pointer to a **SOMDObject Class** object for which to return the **ReferenceData**. This should not be NULL, OBJECT_NIL, an instance of **SOMDClientProxy Class**, or an object reference that was not created using either the **create** or **create_constant** method.

Return Value

The **get_id** method returns a **ReferenceData** structure associated with the referenced object. The caller receives ownership of the *_buffer* member of this structure, and should free it using SOMFree when finished using it.

Example

```
SOM_Scope SOMObject SOMLINK
              somdSOMObjFromRef(SOMPServer somSelf,
                                Environment *ev,
                                SOMDObject objref)
{  SOMObject obj;

   /* test if objref is mine */
   if (!_is_nil(objref, ev) && _somIsA(objref, _SOMDObject) &&
            !_is_proxy(objref, ev) && !_is_SOM_ref(objref, ev)) {
      /* objref was mine, activate persistent object myself */
      ReferenceData rd = _get_id(SOMD_SOMOAObject, ev, objref);
      obj = get_object_from_refdata(ev, &rd);
      SOMFree(rd._buffer);
   } else
     /* it's not one of mine, let parent activate object */
     obj = parent_somdSOMObjFromRef(somSelf, ev, objref);
   return obj;
}
```

Original Class

BOA Class

Related Information

create Method

create_constant Method

dispose Method

somdSOMObjFromRef Method

get_principal Method

Returns the ID of the principal that issued the request.

IDL Syntax

Principal get_principal (
in SOMDObject *obj*,
in Environment* *req_ev*);

Description

The **get_principal** method returns the ID of the principal that issued a request.

Parameters

receiver
A pointer to a BOA (SOMOA) object for the server.

ev
A pointer to the **Environment** structure for the method caller.

obj
A pointer to the object reference that is the target of the method call.

req_ev
A pointer to the **Environment** object passed as input to the request, or the global environment (obtained via **somGetGlobalEnvironment**) if the request was for a method that has no **Environment** parameter.

Return Value

The **get_principal** method returns a pointer to a **Principal** object which identifies the user and host from which a request originated. The caller should not free this object.

Example

```
#include <somd.h>

/* assumed context: inside a method implementation */
void methodBody(SOMObject *somSelf, Environment *ev, ...)
SOM_Scope void  SOMLINK m2(Stack somSelf, Environment *ev,
SOMObject d,

SOMObject c)
{
   Principal p;
   SOMDObject selfRef;
   Environment localev;
   somPrintf("Enter - m2: d = %x c = %x\n", d, c);
   SOM_InitEnvironment(&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);
   somPrintf("userName = %s, hostName = %s\n",
   __get_userName(p, ev),et_hostName(p, ev));
}
```

Original Class

BOA Class

Related Information

Principal Class

impl_is_ready Method

Indicates that a server is ready to process requests.

IDL Syntax

void impl_is_ready (
in ImplementationDef *impl*);

Description

The **impl_is_ready** method indicates that a server is ready to process requests. If the call fails, the server should invoke **activate_impl_failed** before terminating. Otherwise, the server should call **deactivate_impl** prior to termination (normal or abnormal).

When the server invokes **impl_is_ready** on a instance of DSOM's **SOMOA Class**, if the server's **ImplementationDef::config_file** attribute differs from the current SOMENV setting, the contents of the configuration file named by **ImplementationDef::config_file** will be read, any DSOM run-time initialization performed during **SOMD_Init Function** will be refreshed, and for the duration of the server process the setting of **ImplementationDef::config_file** will be prepended to the current SOMENV setting.

A server program should not attempt to export object references or use any other Object Adapter services until it has invoked **impl_is_ready**, as some crucial server initialization steps are performed at that time. The only exception is **SOMOA::activate_impl_failed**.

Parameters

receiver
A pointer to a BOA (SOMOA) object for the server.

ev
A pointer to the **Environment** structure for the method caller.

impl
A pointer to the **ImplementationDef** object indicating which server is ready.

Example

```
/* A server program that takes an ImplId as the first argument */
#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);
   SOMD_SOMOAObject = SOMOANew();
   /* 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);
}
```

Original Class

BOA Class

Related Information

activate_impl_failed Method

deactivate_impl Method

execute_next_request Method

execute_request_loop Method

obj_is_ready Method

obj_is_ready Method

Note: This method is not implemented.

Indicates that an object (server) is ready to process requests.

IDL Syntax

```
void obj_is_ready (
        in SOMDObject obj,
        in ImplementationDef impl);
```

Description

The **obj_is_ready** method is defined by the CORBA 1.1 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”). **obj_is_ready** is meant to be used by unshared servers, to indicate that the object (that is, 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.

Parameters

receiver
A pointer to a BOA (SOMOA) object for the server.

ev
A pointer to the **Environment** structure for the method caller.

obj
A pointer to a **SOMDObject Class** which identifies the object (server) which is ready.

impl
A pointer to the **ImplementationDef** object representing the object that is ready.

Original Class

BOA Class

Related Information

activate_impl_failed Method

deactivate_impl Method

deactivate_obj Method

impl_is_ready Method

set_exception Method

Returns an exception to a client.

IDL Syntax

void set_exception (
in exception_type *major*,
in string *except_name*,
in void* *param*);

Description

The **set_exception** method sets an exception in an **Environment** structure, by calling the **somSetException Function**. The major parameter can have one of three possible values:

NO_EXCEPTION
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
a user-defined exception.

SYSTEM_EXCEPTION
a system-defined exception.

Parameters

receiver
A pointer to a BOA (SOMOA) object for the server.

ev
A pointer to the **Environment** structure for the method caller.

major
An exception types: NO_EXCEPTION, USER_EXCEPTION or SYSTEM_EXCEPTION.

except_name
A **string** representing the exception type identifier.

param
A pointer to the associated data. Ownership of this data structure is passed to the **Environment** and should not be subsequently changed or freed by the caller.

Example

```
#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);
```

Original Class

BOA Class

Context Class

The **Context** class implements the CORBA 1.1 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. They are to be passed as the third parameter to any method whose IDL definition specifies that a context is required.

Note: The Context class is not thread-safe. Multi-threaded applications must be careful that only one thread modifies the state of a given Context object. For example, one thread can not add context property values while another thread is deleting property values.

See the discussion of IDL context expressions in "Context Expression" on page 132 of *Programmer's Guide for SOM and DSOM*.

File Stem

cntxt

Base

SOMObject Class

Metaclass

SOMClass Class

Ancestor Classes

SOMObject Class

New Methods

create_child Method

delete_values Method

destroy Method*

get_values Method

set_one_value Method

set_values Method

* The **destroy** method was defined as **delete** in CORBA 1.1, which conflicts with the **delete** operator in C++. There is a **Context_delete Macro** defined for CORBA 1.1 compatibility.

Overridden Methods

somDefaultInit Method

somDestruct Method

create_child Method

Creates a child of a **Context** object.

IDL Syntax

ORBStatus create_child (
in Identifier *ctx_name*,
out Context *child_ctx*);

Description

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.

Parameters

receiver
A pointer to the **Context** object for which a child is to be created.

ev
A pointer to the **Environment** structure for the method caller.

ctx_name
The name of the child **Context** to be created.

child_ctx
The address where a pointer to the created child **Context** object will be stored. Ownership of the *child_ctx* parameter is transferred to the caller, who is responsible for destroying it using the **destroy Method** (either on the child **Context** object itself or via a call to **Context::destroy** on its parent **Context** using the flag CTX_DELETE_DESCENDANTS).

Return Value

The **create_child** method returns an **ORBStatus** value representing the return code from the operation.

Example

```
#include <somd.h>

Environment ev;
Context cxt, newcxt;
long rc;
SOM_InitEnvironment (&ev);
...
/* 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);
```

Original Class

Context Class

delete_values Method

Deletes property values.

IDL Syntax

ORBStatus delete_values (
in Identifier *prop_name*);

Description

The **delete_values** method deletes the specified property values 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.

Parameters

receiver
A pointer to the **Context** object from which values will be deleted.

ev
A pointer to the **Environment** structure for the method caller.

prop_name
An identifier specifying the property value(s) to be deleted.

Return Value

The **delete_values** method returns an **ORBStatus** value representing the return code from the operation.

Example

```
#include <somd.h>

Environment ev;
Context cxt, newcxt;
long rc;
SOM_InitEnvironment (&ev);
...
/* 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”);
```

Original Class

Context Class

Related Information

get_values Method

set_one_value Method

set_values Method

destroy Method

Deletes a **Context** object.

IDL Syntax

ORBStatus destroy (
in Flags *del_flag*);

Description

The **destroy** method deletes the specified **Context** object.

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 1.1 compatibility, a macro defining **Context_delete** as an alias for **destroy** has been included in the C header files.

Parameters

receiver
A pointer to the **Context** object to be deleted.

ev
A pointer to the **Environment** structure for the method caller.

del_flag
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.

Return Value

The **destroy** method returns an **ORBStatus** value representing the return code from the operation.

Example

```
#include <somd.h>

Environment ev;
Context cxt, newcxt;
long rc;
SOM_InitEnvironment (&ev);
...
/* get the process' default Context */
rc = _get_default_context(SOMD_ORBObject, &ev, &cxt);
/* make newcxt a child Context of 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);
```

Original Class

Context Class

get_values Method

Retrieves the specified property values.

IDL Syntax

ORBStatus get_values (
in Identifier *start_scope*,
in Flags *op_flags*,
in Identifier *prop_name*,
out NVList *values*);

Description

The **get_values** method retrieves the specified **Context** property values. If *prop_name* has a trailing wildcard character(“*”), then all matching properties and their values are returned. OWNERSHIP of the returned **NVList Class** 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.

Parameters

receiver
A pointer to the **Context** object from which the properties are to be retrieved.

ev
A pointer to the **Environment** structure for the method caller.

start_scope
An Identifier specifying the name of the **Context** object at which search for the properties should commence.

op_flags
This parameter supports the CTX_RESTRICT_SCOPE flag; callers also can pass zero.

prop_name
An Identifier specifying the name of the property value(s) to return.

values
The address to store a pointer to the resulting **NVList** object.

Return Value

The **get_values** method returns an **ORBStatus** value representing the return code from the operation.

Example

```
#include <somd.h>

Environment ev;
Context cxt1, cxt2;
string *cxt1props;
long rc, i, numprops;
NVList nvp;
SOM_InitEnvironment (&ev);
...
```

```
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 update cxt2 with that name-value pair */
 if (rc == 0) rc = _set_values(cxt2, &ev, nvp);
 _free(nvp,&ev);
 cxt1props++;
}
```

Original Class

Context Class

Related Information

delete_values Method

set_one_value Method

set_values Method

set_one_value Method

Adds a single property to the specified **Context** object.

IDL Syntax

ORBStatus set_one_value (
in Identifier *prop_name*,
in string *value*);

Description

The **set_one_value** method adds a single property to the specified **Context** object.

Parameters

receiver
A pointer to the **Context** object to which the value is to be added.

ev
A pointer to the **Environment** structure for the method caller.

prop_name
The name of the property to be added. The *prop_name* should not end in an asterisk (*).

value
The value of the property to be added.

Return Value

The **set_one_value** method returns an **ORBStatus** value representing the return code from the operation.

Example

```
#include <somd.h>

Environment ev;
Context cxt, newcxt;
long rc;
SOM_InitEnvironment (&ev);
...
/* get the process' default Context */
rc = _get_default_context(SOMD_ORBObject, &ev, &cxt);
/* make newcxt a child Context of default Context (cxt) */
rc = _create_child(cxt, &ev, "myContext", &newcxt);
rc = _set_one_value(newcxt, &ev, "username", "joe");
```

Original Class

Context Class

Related Information

delete_values Method

get_values Method

set_values Method

set_values Method

Adds or changes one or more property values in the specified **Context** object.

IDL Syntax

ORBStatus set_values (
in NVList *values*);

Description

The **set_values** method sets one or more property values in the specified **Context** object. In the **NVList Class**, the flags field must be set to zero, and the TypeCode field associated with an attribute value must be **TC_string**.

Note: TypeCode constants have the form TC_*typename*. Thus, **TC_string** denotes a string constant in the TypeCode field.

Parameters

receiver
A pointer to the **Context** object for which the properties are to be set.

ev
A pointer to the **Environment** structure for the method caller.

values
A pointer to an **NVList** object containing the properties to be set. The property names in the **NVList** should not end in an asterisk (*).

Return Value

The **set_values** method returns an **ORBStatus** value representing the return code from the operation.

Example

```
#include <somd.h>

Environment ev;
Context cxt1, cxt2;
string *cxt1props;
long rc, i, numprops;
NVList nvp;
SOM_InitEnvironment (&ev);
...
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 update cxt2 with that name-value pair */
 if (rc == 0) rc = _set_values(cxt2, &ev, nvp);
 _free(nvp,&ev);
 cxt1props++;
}
```

Original Class

Context Class

Related Information

delete_values Method

get_values Method

set_one_value Method

ImplementationDef Class

The **ImplementationDef** class defines attributes necessary for the DSOM daemon to find and activate a server and for a server to initialize itself. **ImplementationDef** objects are stored in the Implementation Repository, represented by an object of class **ImplRepository**.

Note: The **ImplementationDef** class is not thread-safe. Multi-threaded applications must be careful that only one thread modifies the attributes of a given **ImplementationDef** object.

File Stem

impldef

Base

CosStream::Streamable

Metaclass

SOMClass Class

Ancestor Classes

SOMObject Class

CosStream::Streamable

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. This identifier is unique throughout the network and can be used as a key into the Implementation Repository. The string returned by the **_get_impl_id** method must be freed by the caller; the input parameter to the **_set_impl_id** method should be freed by the caller if its space was allocated out of the heap.

impl_alias (string)
Contains the “alias” (user-friendly name) for a server implementation, specified by the system administrator when registering the server (such as, via **regimpl**). This alias must be unique within a particular Implementation Repository (and can be used as a key), but is not necessarily unique throughout the network. The string returned by the **_get_impl_alias** method must be freed by the caller; the input parameter to the **_set_impl_alias** method should be freed by the caller if its space was allocated out of the heap.

impl_program (string)
Contains the name of the program or command file to 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. This attribute need not be unique for different **ImplementationDef** objects. For example, many servers are registered to use the DSOM default server program, **somdsvr**.

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, on OS/2

```
icsdebug myserver
```

will start myserver under the control of the icsdebug debugger. Servers that are started automatically by **somdd** will always be passed their **impl_id** as the first parameter.

The string returned by the **_get_impl_program** method must be freed by the caller; the input parameter to the **_set_impl_alias** method should be freed by the caller if its space was allocated out of the heap.

impldef_class (string)
Contains the class name of the implementation definition. Class must inherit from ImplementationDef, which is the default value. The string returned by the _get_impldef_class method must be freed by the caller; the input parameter to the _set_impldef_class method should be freed by the caller if its space was allocated out of the heap.

impl_flags (Flags)
Contains a bit-vector of flags used to identify server options. Currently, the flags are as follows (other flag bits are reserved for use by IBM):

- IMPLDEF_MULTI_THREAD flag indicates that each request should be executed on a separate thread.
- IMPLDEF_IMPLID_SET indicates that the **impl_id** attribute has been set when the **ImplementationDef** object is passed to the **add_impldef Method**. The **ImplRepository::add_impldef** method normally generates the **impl_id** attribute before storing the **ImplementationDef** object in the Implementation Repository. The **add_impdef** method will remove the IMPLDEF_IMPLID_SET flag, if set, prior to storing the **ImplementationDef** object in the Implementation Repository.
- IMPLDEF_SECUREMODE indicates that the server accepts requests from authenticated clients only.

The string returned by the **_get_impl_flags** method must be freed by the caller; the input parameter to the **_set_impl_flags** method should be freed by the caller if its space was allocated out of the heap.

impl_server_class (string)
Contains the name of the **SOMDServer Class** or subclass to be instantiated by the server process, to yield the server’s “server object”. The string returned by the **_get_impl_server_class** method must be freed by the caller; the input parameter to the **_set_impl_server_class** method should be freed by the caller if its space was allocated out of the heap.

config_file (string)
Contains the config file location of the server, if different from the SOMENV setting of the process or shell starting the server (for example, the DSOM daemon). The string returned by the **_get_config_file** method must be freed by the caller; the input parameter to the **_set_config_file** method should be freed by the caller if its space was allocated out of the heap.

Notes

The following table depicts the maximum length of attributes as stored in the Implementation Repository:

Attribute Description	Attribute	Maximum Byte Length
File names	**config_file**	255
Alias names	**impl_alias**	255
Class names	**impldef_class, impl_server_class**	255
Implementation ID	**impl_id**	255
Program name	**impl_program**	255

Table 1. Length of attributes stored in the Implementation Repository

ImplRepository Class

The **ImplRepository** class defines operations necessary to query and update the DSOM Implementation Repository, which is a collection of **ImplementationDef** objects. After a DSOM process invokes **SOMD_Init**, the global variable **SOMD_ImplRepObject** points to the **ImplRepository** object for the process.

In addition to updating the Implementation Repository, methods on ImplRepository also update the Naming Service. Whereas the DSOM daemon and servers refer directly to the Implementation Repository, DSOM clients obtain information about servers from the Naming Service. Hence, when a server is registered (or updated or deleted) in the Implementation Repository, corresponding information is registered in the Naming Service for use by the DSOM Factory Service on behalf of clients.

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.

File Stem

implrep

Base

SOMObject Class

Metaclass

SOMClass Class

Ancestor Classes

SOMObject Class

New Methods

add_class_to_all Method

add_class_to_impldef Method

add_class_with_properties Method

add_impldef Method

delete_impldef Method

find_all_aliases Method

find_all_impldefs Method

find_classes_by_impldef Method,

find_impldef Method

find_impldef_by_alias Method

find_impldef_by_class Method

remove_class_from_all Method

remove_class_from_impldef Method

update_impldef Method

Overridden Methods

somDefaultInit Method

somDestruct Method

add_class_to_all Method

Associates the class name with all **ImplementationDef Class** objects.

IDL Syntax

ORBStatus add_class_to_all (in string *className*);

Description

The **add_class_to_all** method associates the specified *className* with all of the **ImplementationDef** objects currently in the Implementation Repository. (This association does not extend to **ImplementationDef** objects added to the Implementation Repository at a later time.) The new association is stored in the Naming Service for use by the DSOM Factory Service.

Parameters

receiver
A pointer to the **implRepository** object (usually the global variable **SOMD_ImplRepObject**).

env
A pointer to the **Environment** structure for the method caller.

className
A string identifying the class name.

Return Value

A zero is returned to indicate success; otherwise, a DSOM error code is returned.

Example

```
#include <somd.h>
Environment ev;
ORBStatus rc;
SOM_InitEnvironment (&ev);
SOMD_Init(&ev);
rc = _add_class_to_all(SOMD_ImplRepObject, &ev, ”MyClass”);
```

Original Class

ImplRepository Class

add_class_to_impldef Method

Associates a class, identified by name, with a server, identified by its **ImplId**.

IDL Syntax

void add_class_to_impldef (
in ImplId *implid,*
in string *className*);

Description

add_class_to_impldef associates a class with a server that indicates that the server implements the named class. This type of association looks up server implementations via the **find_impldef_by_class Method** and is used by the DSOM Factory Service. The new association is stored in the Naming Service.

The *className* can be a fully scoped class name or an unscoped class name, provided that clients use the same form when requesting a factory for the class using the DSOM Factory Service. Use the fully scoped form when the unscoped form is ambiguous.

For a server to create any class whose DLL can be loaded, use the *className* keyword **_ANY**. For a class that can be instantiated locally within any client process running on the same host, use the *implid* keyword **_LOCAL**.

Parameters

receiver
A pointer to the **ImplRepository** object, usually the variable SOMD_ImplRepObject.

env
A pointer to the **Environment** structure for the method caller.

implid
The **Impl_id** attribute for the **ImplementationDef** of the desired server.

className
A string identifying the class name. The *className* parameter must match the class name as specified when the class was associated with some server (for example, via **regimpl**).

Return Value

An exception is returned if there was an error updating the Naming Service.

Example

```
#include <somd.h>
Environment ev;
...
SOM_InitEnvironment (&ev);
SOMD_Init(&ev);
implid = __get_impl_id(SOMD_ImplDefObject,&ev);
_add_class_to_impldef(SOMD_ImplRepObject,&ev,implid,"Queue");
```

Original Class

ImplRepository Class

add_class_with_properties Method

Associates a class name and specific property and value pairs with an object.

IDL Syntax

ORBStatus add_class_with_properties (
in ImplId *implid*,
in string *className*,
in PVList *pvl*);

Description

add_class_with_properties associates a *className* with an **ImplementationDef** whose **impl_id** attribute is *implid*. This is done to indicate that the server (specified by the **ImplementationDef**) implements the named class. The optional **PVList** sequence can associate the specified properties and values with a class. The new association is stored in the Naming Service.

To indicate that a server can create any class whose DLL can be loaded, use the *className* keyword **_ANY**. To indicate that a class can be instantiated locally within any client process running on the same host, use the *implid* keyword **_LOCAL**.

Parameters

receiver
A pointer to the **ImplRepository** object (usually the global variable SOMD_ImplRepObject).

env
A pointer to the **Environment** structure for the method caller.

implid
The **ImplId** identifier for the **ImplementationDef** of the desired server.

className
A string identifying the class name.

pvl
A sequence of **PVList** containing optional property and value pairs. If no additional properties are to be associated with this class, then this parameter should be NULL.

Return Value

A zero is returned to indicate success; otherwise, a DSOM error code is returned.

Example

```
#include <somd.h>

Environment ev;
ORBStatus rc;
_IDL_SEQUENCE_ImplRepository_PV pvl;
ImplId implid;
ImplementationDef impldef;
SOM_InitEnvironment (&ev);
SOMD_Init(&ev);
...
...
impldef = _find_impldef_by_name(SOMD_ImplRepObject, &ev,
"stackServer");
implid = __get_impl_id(impldef, &ev);
...
```

```
pvl._maximum = 1;
pvl._length = 1;
pvl._buffer = (ImplRepository_PV *)
               SOMMalloc(sizeof(ImplRepository_PV));
pvl._buffer[0].name  = ”PropertyName”;
pvl._buffer[0].value = ”PropertyValue”;
...
rc = _add_class_with_properties(SOMD_ImplRepObject,&ev,implid,
                                ”MyClass”, &pvl);
```

Original Class

ImplRepository Class

add_impldef Method

Adds an implementation definition to the Implementation Repository.

IDL Syntax

void add_impldef (in ImplementationDef *impldef*);

Description

The **add_impldef** method adds the specified **ImplementationDef** object to the Implementation Repository. This is equivalent to registering the server using the **regimpl** tools. In addition to updating the Implementation Repository, **add_impldef** also updates the Naming Service for the DSOM Factory Service. If the Naming Service cannot be updated, if it has not been configured or **somdd** is not running, then an exception will be returned .

All attributes of the input **ImplementationDef** object are optional, with the exception of the **impl_alias**. 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.

Parameters

receiver
A pointer to the **ImplRepository** object (usually the global variable SOMD_ImplRepObject).

env
A pointer to the **Environment** structure for the method caller.

impldef
A pointer to the **ImplementationDef** object to add to the Implementation Repository.

Example

```
#include <somd.h>
Environment ev;
ImplementationDef impldef;
SOM_InitEnvironment (&ev);
SOMD_Init(&ev);
...
impldef = ImplementationDefNew();
__set_impl_program(impldef,&ev,"/u/servers/myserver");
/* set more of the impldef's attributes here */
...
_add_impldef(SOMD_ImplRepObject,&ev,impldef);
```

Original Class

ImplRepository Class

delete_impldef Method

Deletes an implementation definition from the Implementation Repository.

IDL Syntax

void delete_impldef (in ImplId *implid*);

Description

The **delete_impldef** method deletes the specified **ImplementationDef Class** object from the Implementation Repository. In addition to updating the Implementation Repository, **delete_impldef** also updates the Naming Service for use by the DSOM Factory Service. If the Naming Service cannot be updated, if it has not been configured or **somdd** is not running, then an exception will be returned.

Parameters

receiver
A pointer to the **ImplRepository** object (usually the global variable **SOMD_ImplRepObject**).

env
A pointer to the **Environment** structure for the method caller.

implid
The **ImplId** that identifies the server implementation of interest (the **impl_id** attribute of the **ImplementationDef** object to be deleted).

Return Value

An exception is returned if there was an error updating the Implementation Repository.

Example

```
#include <somd.h>
Environment ev;
ImplementationDef impldef;
SOM_InitEnvironment (&ev);
SOMD_Init(&ev);
...
impldef = _find_impldef_by_name(SOMD_ImplRepObject,&ev,
                                "stackServer");
_delete_impldef(SOMD_ImplRepObject,&ev,__get_impl_id(impldef,&ev));
```

Original Class

ImplRepository Class

find_all_aliases Method

Returns a sequence of all server aliases registered in the Implementation Repository.

IDL Syntax

ORBStatus find_all_aliases (out sequence<string> *impl_aliases*);

Description

The **find_all_aliases** method searches the Implementation Repository and returns a sequence containing the **impl_alias** name of each **ImplementationDef** object in it.

Parameters

receiver
A pointer to an object of class **implRepository** (usually the global variable **SOMD_ImplRepObject**).

ev
A pointer to the **Environment** structure for the calling method.

impl_aliases
A sequence containing the **impl_alias** attribute of each **ImplementationDef** object in the Implementation Repository. The structure representing the sequence is created by the caller. The *receiver* allocates storage for the *_buffer* field in the sequence, and the caller is responsible for freeing it.

Return Value

A zero is returned to indicate success; otherwise, a DSOM error code is returned.

Example

```
#include <somd.h>

Environment ev;
sequence <string> implaliases;
SOM_InitEnvironment (&ev);
SOMD_Init(&ev);
find_all_aliases(SOMD_ImplRepObject, &ev, &implaliases);
```

Original Class

ImplRepository Class

find_all_impldefs Method

Returns all the implementation definitions in the Implementation Repository.

IDL Syntax

ORBStatus find_all_impldefs (out sequence<ImplementationDef> *outimpldefs*);

Description

The **find_all_impldefs** method searches the Implementation Repository and returns all the **ImplementationDef** objects in it.

Parameters

receiver
A pointer to an object of class **ImplRepository** (usually the global variable **SOMD_ImplRepObject**).

ev
A pointer to the **Environment** structure for the calling method.

outimpldefs
A sequence of **ImplementationDef** objects is returned. The structure representing the sequence is created by the caller. The *receiver* allocates stroage for the *_buffer* field in the sequence, and the caller is responsible for freeing it.

Return Value

A zero is returned to indicate success; otherwise, a DSOM error code is returned.

Example

```
#include <somd.h>
Environment ev;
sequence <ImplementationDef> impldefs;
SOM_InitEnvironment (&ev);
SOMD_Init(&ev);
. . .
find_all_impldefs(SOMD_ImplRepObject, &ev, &impldefs);
```

Original Class

ImplRepository Class

find_classes_by_impldef Method

Returns a sequence of class names associated with a server.

IDL Syntax

sequence<string> find_classes_by_impldef (in ImplId *implid*);

Description

The **find_classes_by_impldef** method returns the sequence of class names supported by a server with the specified *implid*, as registered via the **regimpl** or using the **add_class_to_impldef Method**.

Parameters

receiver
A pointer to the **ImplRepository** object (usually the global variable **SOMD_ImplRepObject**).

env
A pointer to the **Environment** structure for the method caller.

implid
The **ImplId** that identifies the server implementation of interest (the **impl_id** attribute of the server's **ImplementationDef** object).

Return Value

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.

Example

```
#include <somd.h>

Environment ev;
ImplementationDef impldef;
ImplId implid;
sequence<string> classes;
SOM_InitEnvironment (&ev);
SOMD_Init(&ev);
...
impldef = _find_impldef_by_alias (SOMD_ImplRepObject, &ev,
"stackServer");
implid = __get_impl_id(impldef,&ev);
classes = _find_classes_by_impldef(SOMD_ImplRepObject,&ev,implid);
```

Original Class

ImplRepository Class

find_impldef Method

Returns a server implementation definition given its ID.

IDL Syntax

ImplementationDef find_impldef (in ImplId *implid*);

Description

Finds in the Implementation Repository the **ImplementationDef** object whose **impl_id** attribute is *implid*.

Parameters

receiver
A pointer to the **ImplRepository** object (usually the global variable SOMD_ImplRepObject).

env
A pointer to the **Environment** structure for the method caller.

implid
The **impl_id** attribute of the desired **ImplementationDef.**

Return Value

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.

Example

```
#include <somd.h>

main(int argc, char **argv)
{
  Environment ev;

  SOM_InitEnvironment(&ev);
  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);
  ...
}
```

Original Class

ImplRepository Class

find_impldef_by_alias Method

Returns a server implementation definition given its user-friendly alias.

IDL Syntax

ImplementationDef find_impldef_by_alias (in string *alias_name*);

Description

Finds in the Implementation Repository the **ImplementationDef** object whose **impl_alias** attribute is *alias_name.*

Parameters

receiver
A pointer to the **ImplRepository** object (usually the global variable **SOMD_ImplRepObject**).

env
A pointer to the **Environment** structure for the method caller.

alias_name
User-friendly name used to identify the implementation (the **impl_alias** attribute of the desired **ImplementationDef** object).

Return Value

A copy of the desired **ImplementationDef** object is returned, and ownership of the object is transferred to the caller. Or, if the specified alias is not found in the Implementation Repository, NULL is returned.

An exception is returned if there was an error reading the Implementation Repository.

Example

```
#include <somd.h>

Environment ev;
ImplementationDef impldef;
...
SOM_InitEnvironment (&ev);
SOMD_Init(&ev);
impldef =
   _find_impldef_by_alias(SOMD_ImplRepObject,&ev,"stackServer");
```

Original Class

ImplRepository Class

find_impldef_by_class Method

Returns a sequence of implementation definitions for servers that are associated with a specified class.

IDL Syntax

sequence<ImplementationDef> find_impldef_by_class (in string *className*);

Description

Returns a sequence of **implementationDefs** for those servers that have registered an association with a specified class. Typically, a server is associated with the classes it knows how to implement by registering its known classes via the **add_class_to_impldef Method** or by using the **regimpl** tool.

Parameters

receiver
A pointer to the **implRepository** object (usually the global variable **SOMD_ImplRepObject**).

env
A pointer to the **Environment** structure for the method caller.

className
A string whose value is the class name of interest.

Return Value

Copies of all **ImplementationDef** objects are returned in a sequence. Ownership of the sequence structure, the object array buffer, and the objects themselves is transferred to the caller.

An exception is returned if there was an error reading the Implementation Repository.

Example

```
#include <somd.h>

Environment ev;
sequence<ImplementationDef> impldefs;
...
SOM_InitEnvironment [&ev];
SOMD_Init[&ev];
impldefs = _find_impldef_by_class(SOMD_ImplRepObject,&ev,"Stack");
```

Original Class

ImplRepository Class

remove_class_from_all Method

Removes the association of a particular class from all servers.

IDL Syntax

void remove_class_from_all (in string *className*);

Description

The **remove_class_from_all** method removes the association of a particular class with all servers currently registered in the Implementation Repository. Typically, a server is associated with the classes it knows how to implement by registering its known classes via the **add_class_to_impldef Method** or by using the **regimpl** tool.

remove_class_from_all also removes the association from the Naming Service. If the Naming Service cannot be updated, if it has not be configured using **som_cfg**, or if **somdd** is not running, then an exception will be returned but the Implementation Repository will still be updated.

Parameters

receiver
A pointer to an object of class **ImplRepository** (usually the global variable **SOMD_ImplRepObject**).

ev
A pointer to the **Environment** structure for the calling method.

className
A string whose value is the class name of interest.

Example

```
#include <somd.h>

Environment ev;
...
SOM_InitEnvironment (&ev);
SOMD_Init(&ev);
remove_class_from_all(SOMD_ImplRepObject, &ev, "Stack");
```

Original Class

ImplRepository Class

remove_class_from_impldef Method

Removes the association of a particular class with a server.

IDL Syntax

void remove_class_from_impldef (
in ImplId *implid,*
in string *className*);

Description

Removes the specified class name from the set of class names associated with the server implementation identified by *implid.* Typically, a server is associated with the classes it knows how to implement by registering its known classes via the **add_class_to_impldef Method** or by using the **regimpl** tool.

Parameters

receiver
A pointer to the **ImplRepository** object (usually the global variable **SOMD_ImplRepObject**).

env
A pointer to the **Environment** structure for the method caller.

implid
The **impl_id** attribute of the **ImplementationDef** object of the desired server.

className
A string whose value is the class name of interest.

Return Value

An exception is returned if there was an error updating the Implementation Repository.

Example

```
#include <somd.h>
Environment ev;
ImplementationDef impldef;
ImplId implid;
...
SOM_InitEnvironment (&ev);
SOMD_Init(&ev);
impldef = _find_impldef_by_alias (SOMD_ImplRepObject, &ev,
"stackServer");
implid = __get_impl_id(impldef,&ev);
_remove_class_from_impldef(SOMD_ImplRepObject,&ev,implid,"Queue");
```

Original Class

ImplRepository Class

update_impldef Method

Updates an implementation definition in the Implementation Repository.

IDL Syntax

void update_impldef (in ImplementationDef *impldef*);

Description

The **update_impldef** method replaces the state of the specified **ImplementationDef** object in the Implementation Repository. The **impl_id** attribute of *impldef* determines which object gets updated in the Implementation Repository. In addition to updating the Implementation Repository, **update_impldef** also updates the Naming Service for use by the DSOM Factory Service. If the Naming Service cannot be updated, if it has not been been configured or **somdd** is not running, then an exception will be returned. However, the Implementation Repository will still be updated provided no information was previously stored in the Naming Service for the given **ImplementationDef**. The next time the server's Implementation Repository entry is updated, DSOM will attempt to update the Naming Service.

Parameters

receiver
A pointer to the **implRepository** object (usually the global variable **SOMD_ImplRepObject**).

env
A pointer to the **Environment** structure for the method caller.

impldef
A pointer to an **ImplementationDef** object, whose values are to be updated in the Implementation Repository.

Return Value

An exception is returned if there was an error updating the Implementation Repository.

Example

```
#include <somd.h>
#include <implrep.h>

Environment ev;
ImplementationDef impldef;
...
SOM_InitEnvironment (&ev);
SOMD_Init(&ev);
impldef = _find_impldef_by_alias (SOMD_ImplRepObject, &ev,
"stackServer") ;
__set_impl_program(impldef,&ev,"/u/joe/bin/myserver");
_update_impldef(ir,&ev,impldef);
```

Original Class

ImplRepository Class

NVList Class

Description

The type **NamedValue** is a standard data type defined in CORBA (see **somdtype.idl** or the CORBA 1.1 specification, page 106). **NamedValue** 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.

Note: The NVList class is not thread-safe. Multi-threaded applications must be careful that only one thread modifies the state of a given NVList object. For example, one thread cannot add items to an NVList while another thread is attempting to free it.

Additional information about **NVList** is contained in "Dynamic Invocation Interface" on page 311 of *Programmer's Guide for SOM and DSOM* and in Chapter 6 of the CORBA 1.1 specification.

File Stem

nvlist

Base

SOMObject Class

Metaclass

SOMClass Class

Ancestor Classes

SOMObject Class

New Methods

add_item Method

free Method

free_memory Method

get_count Method

get_item Method*

set_item Method*

(* These methods were added by DSOM to supplement CORBA 1.1 methods.)

Overridden Methods

somDefaultInit Method

somDestruct Method

add_item Method

Adds an item to the specified **NVList**.

IDL Syntax

ORBStatus add_item (
in Identifier *item_name*,
in TypeCode *item_type*,
in void* *value*,
in long *value_len*,
in Flags *item_flags*);

Description

The **add_item** method adds an item to the end of the specified list.

Parameters

receiver
A pointer to the **NVList** object to which the item will be added.

env
A pointer to the **Environment** structure for the method caller.

item_name
The name of the item to be added.

item_type
The data type of the item to be added.

value
A pointer to the value of the item to be added. This value is not copied unless *item_flags* is set to IN_COPY_VALUE.

value_len
The length of the item value to be added.

item_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 addition, *item_flags* may also contain the following values:

IN_COPY_VALUE An internal copy of the argument is made and used. (Currently DSOM has the limitation that only a shallow copy is made.)

DEPENDENT_LIST Indicates that a specified sublist must be freed when the parent list is freed.

Return Value

The **add_item** method returns an **ORBStatus** value representing the return code from the operation.

Example

```
#include <somd.h>

Environment ev;
NVList plist;
ORBStatus rc;
...
SOM_InitEnvironment (&ev);
SOMD_Init(&ev);
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);
```

Original Class

NVList Class

Related Information

create_list Method

free Method

free_memory Method

get_count Method

get_item Method

set_item Method

free Method

Frees a specified **NVList.**

IDL Syntax

ORBStatus free ( );

Description

The free method frees a n **NVList** object and any associated memory. It makes an implicit call to the **free_memory** method.

Parameters

receiver
A pointer to the **NVList** object to be freed.

env
A pointer to the **Environment** structure for the method caller.

Return Value

The method returns an **ORBStatus** value representing the return code from the operation.

Example

```
#include <somd.h>
Environment ev;
long nargs;
NVList arglist;
ORBStatus rc;
...
SOM_InitEnvironment (&ev);
SOMD_Init(&ev);
rc = _create_list(SOMD_ORBObject, &ev, nargs, &arglist);
...
rc= _free(arglist,&ev);
```

Original Class

NVList Class

Related Information

ORBfree Function

free_memory Method

free_memory Method

Frees any dynamically allocated out-arg memory associated with the specified list.

IDL Syntax

ORBStatus free_memory ( );

Description

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.**

Parameters

receiver
A pointer to the **NVList** object whose out-arg memory is to be freed.

env
A pointer to the **Environment** structure for the method caller.

Return Value

The **free_memory** method returns an **ORBStatus** value representing the return code from the operation.

Example

```
#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 */
SOM_InitEnvironment (&ev);
SOMD_Init(&ev);
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 */
}
```

Original Class

NVList Class

Related Information

ORBfree Function

free Method

get_count Method

Returns the total number of items allocated for a list.

IDL Syntax

ORBStatus get_count (out long *count*);

Description

The **get_count** method returns the total number of allocated items in the specified list.

Parameters

receiver
A pointer to the **NVList** object on which count is desired.

env
A pointer to the **Environment** structure for the method caller.

count
A pointer to where the method will store the **long** integer count value.

Return Value

The **get_count** method returns an **ORBStatus** value representing the return code from the operation.

Example

```
#include <somd.h>

Environment ev;
long nargs, list_size;
NVList arglist;
ORBStatus rc;

...
SOM_InitEnvironment (&ev);
SOMD_Init(&ev);
rc = _create_list(SOMD_ORBObject, &ev, nargs, &arglist);
...
rc = _get_count(arglist,&ev,&list_size);
```

Original Class

NVList Class

Related Information

add_item Method

create_list Method

get_item Method

set_item Method

get_item Method

Returns the contents of a specified list item.

IDL Syntax

ORBStatus get_item (
in long *item_number*,
out Identifier *item_name*,
out TypeCode *item_type*,
out void* *value*,
out long *value_len*,
out Flags *item_flags*);

Description

The **get_item** method gets an item from the specified list. Items are numbered beginning at zero. Ownership of all the **out** parameters is not transferred to the caller.

Parameters

receiver
A pointer to an **NVList** object.

env
A pointer to the **Environment** structure for the method caller.

item_number
The position (index) of the desired 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
A pointer to where the name of the item should be returned.

item_type
A pointer to where the data type of the item should be returned.

value
A pointer to where a pointer to the value of the item should be returned.

value_len
A pointer to where the length of the item value should be returned.

item_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.

In addition, *item_flags* can have the following values:

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. (This setting is not currently supported.)

Return Value

Returns 0 for success, or a DSOM error code for failure (often because *item_number*+1 exceeds the number of items in the list).

Example

```
#include <somd.h>

Environment ev;
long i, nArgs;
ORBStatus rc;
Identifier name;
TypeCode typeCode;
void *value;
long len;
Flags flags;
NVList argList;

SOM_InitEnvironment (&ev);
SOMD_Init(&ev);
/* get number of args */
rc = _get_count(argList, &ev, &nArgs);
for (i = 0; i < nArgs; i++) {
/* get item description */
rc = _get_item(argList, &ev, i, &name, &typeCode, &value, &len,
&flags);
}
```

Original Class

NVList Class

Related Information

add_item Method

create_list Method

set_item Method

set_item Method

Sets the contents of an item in a list.

IDL Syntax

ORBStatus set_item (
in long *item_number*,
in Identifier *item_name*,
in TypeCode *item_type*,
in void* *value*,
in long *value_len*,
in Flags *item_flags*);

Description

The **set_item** method sets the contents of an item in the list.

Parameters

receiver
An **NVList** object.

ev
The **Environment** structure for the method caller.

item_number
The position (index) of the desired 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
The name of the item.

item_type
The data type of the item.

value
The value of the item.

value_len
The length of the item value.

item_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.

In addition, *item_flags* can have the following values:

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. (This setting is not currently supported.)

Return Value

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).

Example

```
#include <somd.h>

Environment ev;
long i, nArgs;
ORBStatus rc;
Identifier name;
TypeCode typeCode;
void *value;
long len;
Flags flags;
NVList argList;

SOM_InitEnvironment (&ev);
SOMD_Init(&ev);
/* get number of args */
rc = _get_count(argList, &ev, &nArgs);
for (i = 0; i < nArgs; i++)
{
/* change item description */
rc = _set_item(argList, &ev,i, name, typeCode, value, len, flags);
}
```

Original Class

NVList Class

Related Information

add_item Method

create_list Method

get_item Method

ObjectMgr Class

All methods of this class have been deprecated. Use of this class is discouraged. While the methods of this class are supported in the current release of SOMobjects, IBM may remove this class from a subsequent release.

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 (that is, release any local memory associated with the object without necessarily destroying the object itself), or
- Destroy an object.

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.

File Stem

om

Base

SOMObject Class

Metaclass

SOMMSingleInstance Metaclass

Ancestor Classes

SOMObject Class

Subclasses

SOMDObjectMgr Class

Deprecated Methods

IBM discourages using the following **ObjectMgr** methods. While these methods still are supported in this release of SOMobjects, IBM may remove these methods from subsequent releases.

somdDestroyObject*

somdGetIdFromObject*

somdGetObjectFromId*

somdNewObject*

somdReleaseObject*

(* This class and its methods were added by DSOM to supplement the published CORBA 1.1 interfaces.)

ORB Class

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** defines operations used by the Dynamic Invocation Interface for creating lists and determining the default context. In addition, **ORB** provides initialization methods that list and retrieve references to basic object services.

After a DSOM process invokes **SOMD_Init Function**, the global variable **SOMD_ORBObject** points to the single instance of **ORB** for that process.

File Stem

orb

Base

SOMObject Class

Metaclass

SOMMSingleInstance Metaclass

Ancestor Classes

SOMObject Class

New Methods

create_list Method

create_operation_list Method

get_default_context Method

list_initial_services Method

object_to_string Method

resolve_initial_references Method

string_to_object Method

create_list Method

Creates an **NVList** of the specified size.

IDL Syntax

ORBStatus create_list (
in long *count*,
out NVList *new_list*);

Description

Creates an **NVList** list of the specified size, typically for use in **Requests**. Ownership of the allocated *new_list* is transferred to the caller.

Parameters

receiver
A pointer to the **ORB** object (referred to by the global variable **SOMD_ORBObject**).

env
A pointer to the **Environment** structure for the method caller.

count
An integer representing the number of elements to allocate for the list.

new_list
A pointer to the address where the method will store a pointer to the allocated **NVList** object.

Return Value

Returns an **ORBStatus** value representing the return code of the operation.

Example

```
#include <somd.h>

Environment ev;
long nargs = 5;
NVList arglist;
ORBStatus rc;
...
SOM_InitEnvironment (&ev);
SOMD_Init(&ev);
rc = _create_list(SOMD_ORBObject, &ev, nargs, &arglist);
```

Original Class

ORB Class

Related Information

create_operation_list Method

NVList Class

Request Class

create_operation_list Method

Creates an **NVList** initialized with the argument descriptions for a given operation.

IDL Syntax

ORBStatus create_operation_list (
in OperationDef *oper*,
out NVList *new_list*);

Description

Creates an **NVList** list for the specified operation, for use in **Requests** invoking that operation. Ownership of the allocated *new_list* is transferred to the caller.

Parameters

receiver
A pointer to the **ORB** object (referred to by the global variable **SOMD_ORBObject**).

env
A pointer to the **Environment** structure for the method caller.

oper
A pointer to the **OperationDef** object representing the operation for which the **NVList** is to be initialized, looked up in the Interface Repository.

new_list
A pointer to where the method will store a pointer to the resulting argument list.

Return Value

Returns an **ORBStatus** value representing the return code of the operation.

Example

```
#include <somd.h>

Environment ev;
OperationDef opdef;
NVList arglist;
long rc;
SOM_InitEnvironment (&ev);
SOMD_Init(&ev);
/* 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);
```

Original Class

ORB Class

Related Information

create_list Method

NVList Class

Repository Class

Request Class

get_default_context Method

Returns the default process **Context** object.

IDL Syntax

ORBStatus get_default_context (out Context *ctx*);

Description

The **get_default_context** method gets the default process **Context** object. Ownership of the allocated **Context** object is transferred to the caller.

Parameters

receiver
A pointer to the **ORB** object (referred to by the global variable SOMD_ORBObject).

env
A pointer to the **Environment** structure for the method caller.

ctx
A pointer to where the method will store a pointer to the returned **Context** object.

Return Value

Returns an **ORBStatus** return code: 0 indicates success, while a non-zero value is a DSOM error code (see “Error Codes” on page 399 of *Programmer’s Guide for SOM and DSOM*).

Example

```
#include <somd.h>

Environment ev;
Context cxt;
long rc;
...
SOM_InitEnvironment (&ev);
SOMD_Init(&ev);
rc = _get_default_context(SOMD_ORBObject, &ev, &cxt);
```

Original Class

ORB Class

list_initial_services Method

Lists the run-time objects available by calling the **resolve_initial_references** method.

IDL Syntax

ObjectIdList list_initial_services ();

Description

Returns a list of well-known strings (**ObjectId**s) that each specifies an object service available by calling **ORB::resolve_initial_references.** Ownership of allocated memory is transferred to the caller. The caller should free the *_buffer* of the returned sequence and each of the strings contained in the *_buffer* using the **SOMFree Function**. Three **ObjectId**s are defined: InterfaceRepository, NameService and FactoryService.

Parameters

receiver
A pointer to an object of class ORB (referred to by the global variable **SOMD_ORBObject**.)

ev
A pointer to the **Environment** structure for the calling method.

Return Value

Returns a sequence of strings.

Example

```
#include <somd.h>

Environment ev;
ObjectIdList services;
int i;
SOM_InitEnvironment (&ev);
SOMD_Init(&ev);
services = _list_initial_services(SOMD_ORBObject, &ev);
```

Original Class

ORB Class

Related Information

resolve_initial_references Method

object_to_string Method

Converts an object reference to an external form that can be stored outside the ORB.

IDL Syntax

string object_to_string (in SOMObject *obj*);

Description

The **object_to_string** method converts the object reference to a form (string) which can be stored externally. This string can then be passed to **string_to_object** to recover the original object reference.

If the object reference is a local object, rather than a proxy, and the calling process has server capability, then the **somdRefFromSOMObj Method** will be invoked on the server object to generate a **SOMDObject Class** corresponding to the local object; that **SOMDObject** will then be converted to string form. The resulting string form can be passed to **string_to_object** either to retrieve the pointer to the original local object if invoked from within the same server process or to construct a valid proxy object for the object .

If the object reference is a local object, rather than a proxy object, but the calling process does not have server capability, then the result of **object_to_string** is valid only within the calling process for the duration of that process and as long as the input object resides in the address space of that process.

Ownership of allocated memory is transferred to the caller. The caller should free the result using the **SOMFree Function**.

Note: In DSOM 2.x, **string_to_object** invocation within the server process in which the object resided, returned a **SOMDObject** rather than the object. **object_to_string** required an input **SOMDObject** rather than a **SOMObject Class**. In this release, to increase local and remote transparency, **string_to_object** and **object_to_string** map between a **SOMObject** and the string form of a reference to that object. Server code does not need to invoke **SOMDServer::somdRefFromSOMObj** before **object_to_string** nor invoke **SOMDServer::somdSOMObjFromRef** after **string_to_object**.

Parameters

receiver
A pointer to the **ORB** object (referred to by the SOMD_ORBObject global variable).

env
A pointer to the **Environment** structure for the method caller.

obj
A pointer to a **SOMObject** object representing the reference to be converted. This can be either a local or a proxy object.

Return Value

Returns a string representing the external form of the referenced object.

Example

```
#include <somd.h>
#include <car.h>
Environment ev;
Car car;
string objrefstr;
SOM_InitEnvironment(&ev);
```

```
SOMD_Init(&ev);
/* create a remote Car object */
car = somdCreate(&ev, "Car", TRUE);
/* save the reference to the object */
objrefstr = _object_to_string(SOMD_ORBObject, &ev, car);
FileWrite("/u/joe/mycar", objrefstr);
SOMFree(objrefstr);
```

Original Class

ORB Class

Related Information

string_to_object Method

resolve_initial_references Method

Returns an object reference for the requested object service.

IDL Syntax

SOMObject resolve_initial_references (
in ObjectId ***identifier*****)**
raises (***InvalidName*****);**

Description

resolve_initial_references takes a single **ObjectId** and returns an appropriate object reference for the requested object service. This method may be called during client initialization to get a handle to a basic run-time object. **ObjectIds** can be retrieved by calling **ORB::list_initial_services**. There are three **ObjectIds**: InterfaceRepository, NameService and FactoryService. Calling with **ObjectId** set as InterfaceRepository returns an object of type **Repository**, a local instance of the Interface Repository; as NameService, type **ExtendedNaming_ExtendedNamingContext**, the root context of the local naming tree; and as FactoryService**,** type **ExtendedNaming_ExtendedNamingContext**, the naming context where SOM object factories are stored.

Ownership of the returned object is transferred to the caller who should free the result using **somFree Method**.

Parameters

receiver
A pointer to an object of class **ORB**.

ev
A pointer to the **Environment** structure for the calling method.

identifier
A string representing a basic object service:

"InterfaceRepository" Returns an object of type Repository, a local instance of the Interface Repository.

"NameService" Returns an item of type **ExtendedNaming::ExtendedNamingContext**, the root context of the local naming tree

"FactoryService" Returns an item of type **ExtendedNaming::ExtendedNamingContext**, the naming context where SOM object factories are stored.

Return Value

It must be cast to the actual object class. If an exception occurs, OBJECT_NIL is returned.

Example

```
#include <somd.h>
#include <repostry.h>

Environment ev;
Repository repo;
SOM_InitEnvironment (&ev);
SOMD_Init(&ev);
repo = (Repository) _resolve_initial_references(
     SOMD_ORBObject, &ev, "InterfaceRepository");
```

Original Class

ORB Class

Related Information

list_initial_services Method

string_to_object Method

Converts an externalized form of an object reference into an object reference.

IDL Syntax

SOMObject string_to_object (in string *str*);

Description

The **string_to_object** method converts the externalized form of an object reference produced by **object_to_string** into an object reference.

Note: In DSOM 2.x, **string_to_object** invocation within the object server process returned a SOMDobject rather than the object. **object_to_string** required an input SOMDObject rather than a SOMObject. In this release **string_to_object** and **object_to_string** map between a SOMObject and the string form of the reference. Server code does not need to invoke **somdRefFromSOMObj Method** before **object_to_string** nor **somdSOMObjFromRef** after **string_to_object**.

The caller should invoke the **release Method** on this object reference when the calling process finishes.

Parameters

receiver
A pointer to the ORB object (referred to by the global variable SOMD_ORBObject).

env
A pointer to the **Environment** structure for the method caller.

str
A pointer to a character string representing the externalized form of the object reference.

Return Value

Returns a SOMobject object.

Example

```
#include <somd.h>
#include <car.h>

Environment ev;
Car car;
string objrefstr;
...
SOM_InitEnvironment (&ev);
SOMD_Init(&ev);
/* restore proxy from its string form */
FileRead("/u/joe/mycar", &objrefstr);
car = _string_to_object(SOMD_ORBObject, &ev, objrefstr);
```

Original Class

ORB Class

Related Information

object_to_string Method

Principal Class

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 within a server.

A Principal object is returned by the **SOMOA::get_principal** method. The parameters of the **get_principal** method identify the environment and target object associated with a particular request. The object adapter uses this information to create a Principal object that identifies the caller.

Note: The Principal class is not thread-safe. However, users should not set the *userName* and *hostName* attributes. The DSOM run time will properly initialize these values.

Details of the Principal object are not defined in the CORBA 1.1 specification; the attributes defined are required by DSOM.

File Stem

principal

Base

SOMObject Class

Metaclass

SOMClass Class

Ancestor Classes

SOMObject Class

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. The value of the *userName* attribute is the user name with which the client logged in on the client's machine. If the user has not logged in (or if **LOGIN_INFO_SOURCE** is not set in the SOMobjects configuration file), the user is treated as an unauthenticated user and the *userName* attribute will be an empty string (""). The result of calling **_get_userName** should not be freed by the caller.

hostName (string)
Identifies the name of the host from where the request originated. Currently, this value is obtained from the **HOSTNAME** environment variable or the **HOSTNAME** setting in the [somd] stanza of the configuration file in the process that invoked the request. If, however, the client is not authenticated, the *hostName* attribute will be set to LOCALHOST. The result of calling **_get_hostName** should not be freed by the caller.

Overridden Methods

somDefaultInit Method

somDestruct Method

Request Class

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**.

Note: The Request class is not thread-safe. A Request object should not be modified once it is initialized. Multi-threaded applications must be written to ensure that Request objects are used properly. For example, it is invalid to invoke on a Request object, then change the Request object before receiving the response to the initial invocation.

File Stem

request

Base

SOMObject Class

Metaclass

SOMClass Class

Ancestor Classes

SOMObject Class

New Methods

add_arg Method

destroy Method*

get_response Method

invoke Method

send Method

(* 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

somDefaultInit Method

somDestruct Method

add_arg Method

Incrementally adds an argument to a **Request** object.

IDL Syntax

ORBStatus add_arg (
in Identifier *name*,
in TypeCode *arg_type*,
in void* *value*,
in long *len*,
in Flags *arg_flags*);

Description

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.

Parameters

receiver
A pointer to a **Request** object.

ev
A pointer to the **Environment** structure for the method caller.

name
An identifier representing the name of the argument to be added.

arg_type
The typecode for the argument to be added.

value
A pointer to the argument value to be added.

len
The length of the argument.

arg_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.

ARG_INOUT The argument is input/output.

In addition, *arg_flags* can have the following values:

IN_COPY_VALUE Indicates a copy of the *value* is stored in this **Request** object. This flag is ignored for INOUT and OUT arguments.

DEPENDENT_LIST Not currently supported.

Return Value

The **add_arg** method returns an **ORBStatus** value representing the return code of the operation.

Example

```
#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 this 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;
SOM_InitEnvironment (&ev);
SOMD_Init(&ev);

/* 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);
```

Original Class

Request Class

destroy Method

Deletes the memory allocated by the ORB for a **Request** object.

IDL Syntax

ORBStatus destroy ( );

Description

destroy deletes the **Request** object and all associated memory. **destroy** is the same as the **delete** method in the CORBA 1.1 specification. However, the word "delete" is a reserved operator in C++. For CORBA compatibility, **Request_delete Macro**, an alias for **destroy**, has been included in the C header files.

Parameters

receiver
A pointer to a **Request** object.

ev
A pointer to the **Environment** structure for the method caller.

Return Value

The **destroy** method returns an **ORBStatus** value representing the return code.

Example

```
#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 this 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
*/
SOM_InitEnvironment (&ev);
SOMD_Init(&ev);
/* 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);
rc = _get_response(reqObj, &ev, (Flags)0);
/* use the result */
if (result->argument._value == 9600) {...}
/* throw away the reqObj */
_destroy(reqObj, &ev);
```

Original Class

Request Class

Related Information

get_response Method

invoke Method

send Method

get_response Method

Determines whether an asynchronous **Request** has completed.

IDL Syntax

ORBStatus get_response (in Flags *response_flags*);

Description

The **get_response** method determines whether the asynchronous **Request** has completed.

Parameters

receiver
A pointer to a **Request** object.

ev
A pointer to the **Environment** structure for the method caller.

response_flags
A Flags bitmask (unsigned long) containing control information for the **get_response** method. The *response_flags* argument may have the value RESP_NO_WAIT that Indicates the caller does not want to wait for a response.

Return Value

The **get_response** method returns an ORBStatus value representing the return code of the operation.

Example

```
#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 this 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.*/
SOM_InitEnvironment (&ev);
SOMD_Init(&ev);
/* 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) {...}
```

Original Class

Request Class

Related Information

invoke Method

send Method

Request_delete Macro

invoke Method

Invokes a **Request** synchronously, waiting for the response.

IDL Syntax

ORBStatus invoke (in Flags *invoke_flags*);

Description

The **invoke** method sends a **Request** synchronously, waiting for the response.

Parameters

receiver
A pointer to a **Request** object.

ev
A pointer to the **Environment** structure for the method caller.

invoke_flags
A Flags bitmask (unsigned long) representing control information for the **invoke** method. There are currently no flags defined for the **invoke** method.

Return Value

Returns an **ORBStatus** value representing the return code of the operation.

```
#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;
SOM_InitEnvironment (&ev);
SOMD_Init(&ev);
/* 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);
/* 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);
```

Original Class

Request Class

Related Information

get_response Method

send Method

Request_delete Macro

send Method

Invokes a **Request** asynchronously.

IDL Syntax

ORBStatus send (in Flags *invoke_flags*);

Description

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**.

Parameters

receiver
A pointer to a **Request** object.

ev
A pointer to the **Environment** structure for the method caller. This environment structure is used only to record system exceptions encountered during the sending of the request. Exceptions raised by the target method invocation, and system exceptions raised during the receiving of the response, are stored in the **Environment** structure provided to the subsequent **get_response** method.

invoke_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.

Return Value

Returns an **ORBStatus** value representing the return code from the operation.

Example

```
#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
*/
SOM_InitEnvironment (&ev);
SOMD_Init(&ev);
/* 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);
```

Original Class

Request Class

Related Information

get_response Method

invoke Method

Request_delete Macro

SOMDClientProxy Class

The **SOMDClientProxy** class implements DSOM proxy objects in clients. 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 (but not the implementation) are inherited from the target class.

SOMDClientProxy inherits from the metaclass **SOMMProxyForObject**, which is a base class for creating proxies. Every proxy contains the sommProxyDispatch method, inherited from **SOMMProxyForObject**. This method is used to dynamically dispatch a method on an object, and it can also be overridden with application-specific dispatching mechanisms. In **SOMDClientProxy**, the **sommProxyDispatch** method is overridden to forward method calls to the corresponding remote target object. (For more information, see “SOMMProxyForObject Class” on page 368. For additional information and an example, see “Creating User-Supplied Proxies” on page 319 of *SOMobjects Developer’s Toolkit Programmer’s Guide*.

Note: The **SOMDClientProxy** class is thread-safe. Multiple threads may simultaneously send requests and receive responses on the same proxy. It is still the class implementor’s responsibility to ensure the thread-safety of the target object. Also, one thread should not change the state of the proxy (for example, by calling **somdReleaseResources** or **somdProxyFree**) while other threads are still using the proxy.

Almost all methods invoked on a default proxy are simply forwarded and invoked on the remote object. This is true for all methods introduced by the target class. However, some methods introduced by **SOMDClientProxy** have special behavior. A number of methods are not forwarded to the remote object because their definition makes more sense in the local context.

Following is a complete list of methods that are executed on the local proxy:

create_request	**somdReleaseResources**
create_request_args	**somDumpSelf**
duplicate	**somDumpSelfInt**
is_proxy	**somGetSize**
release	**somIsA**
somClassDispatch	**somIsInstanceOf**
somdProxyGetClass	**somPrintSelf**
somdProxyGetClassName	**somRespondsTo**

SOMDClientProxy introduces a few methods that forward other methods to the remote object:

- **somdTargetFree** invokes **somFree** on the target object.
- **somdTargetGetClass** invokes **somGetClass** on the target object.
- **somdTargetGetClassName** invokes **somGetClassName** on the target object.

A small number of methods execute both on the proxy and the remote object. Most of these methods deal with proxy destruction, as follows:

somDefaultInit Method
If the proxy has not been initialized yet, **somDefaultInit** initializes the proxy. If the proxy is already initialized, **somDefaultInit** is forwarded to the target object. (Proxy objects that DSOM creates will be initialized upon creation automatically.)

somDestruct Method
If the proxy is still initialized, **somDestruct** is forwarded to the target object, then the proxy is uninitialized and destroyed. If the proxy is no longer initialized, **somDestruct** destroys the proxy.

somFree Method
Invokes **somFree** on the target object then calls **release** to uninitialize and destroy the proxy.

File Stem

somdcprx

Base

SOMDObject Class

Metaclass

SOMClass Class

Ancestor Classes

SOMMProxyForObject Class

SOMDObject Class

New Methods

somdProxyGetClass Method*

somdProxyGetClassName Method*

somdReleaseResources Method*

somdTargetFree Method*

somdTargetGetClass Method*

somdTargetGetClassName Method*

(* This class and its methods were added by DSOM to supplement the published CORBA 1.1 interfaces.)

Deprecated Method

somdProxyFree*

Overridden Methods

create_request Method

create_request_args Method

duplicate Method

is_proxy Method

is_SOM_ref Method

release Method

somDefaultInit Method

somDestruct Method

som(Class)Dispatch Method

somDumpSelf Method

somDumpSelfInt Method

somFree Method

sommProxyDispatch Method

somPrintSelf Method

somdProxyGetClass Method

Returns the class object for the local proxy object.

IDL Syntax

SOMClass somdProxyGetClass ( );

Description

The **somdProxyGetClass** method returns a pointer to the proxy's class object. This method has been provided for when the application program wants to be explicit about getting the class object for the proxy object versus the target object.

Parameters

receiver
A pointer to the **SOMDClientProxy** object. When invoking this method on an object using the C++ bindings, if the object has not been declared to be of type **SOMDClientProxy** (or some derived type), then it is necessary to explicitly cast the object to **SOMDClientProxy**.

ev
A pointer to the **Environment** structure for the method caller.

Return Value

The **somdProxyGetClass** method returns a pointer to the class object for the local proxy object.

Example

```
#include <somd.h>
#include <car.h>

Environment ev;
Car car;
SOMClass carProxyClass;
string somdObjectId;
...
SOM_InitEnvironment (&ev);
SOMD_Init(&ev);
/* restore proxy from its string form */
FileRead("/u/joe/mycar", &somdObjectId);
car = _string_to_object(SOMD_ORBObject, &ev, somdObjectId);
...
carProxyClass = _somdProxyGetClass(car, &ev);
```

Original Class

SOMDClientProxy Class

somdProxyGetClassName Method

Returns the class name for the local proxy object.

IDL Syntax

string somdProxyGetClassName ( );

Description

The **somdProxyGetClassName** method returns the proxy's class name. This method has been provided for when the application program wants to be explicit about getting the class name of the proxy object versus the target object.

Parameters

receiver
A pointer to the **SOMDClientProxy** object for the desired remote target object. When invoking this method on an object using the C++ bindings, if the object has not been declared to be of type **SOMDClientProxy** (or some derived type), then it is necessary to explicitly cast the object to **SOMDClientProxy)**.

ev
A pointer to the **Environment** structure for the method caller.

Return Value

The **somdProxyGetClassName** method returns a string containing the class name of the local proxy object. This string should not be freed by the caller.

Example

```
#include <somd.h>
#include <car.h>

Environment ev;
Car car;
string carProxyClassName;
string somdObjectId;
...
SOM_InitEnvironment (&ev);
SOMD_Init(&ev);
/* restore proxy from its string form */
FileRead("/u/joe/mycar", &somdObjectId);
car = _string_to_object(SOMD_ORBObject, &ev, somdObjectId);
...
carProxyClassName = _somdProxyGetClassName(car, &ev);
```

Original Class

SOMDClientProxy Class

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.

IDL Syntax

void somdReleaseResources ( );

Description

somdReleaseResources instructs a proxy object to release any memory it is holding from a remote method invocation where a parameter or result was designated as object-owned.

When a DSOM client program makes a remote invocation via a proxy, and the invocation has an object-owned parameter or return result, the client-side memory of the parameter/result is owned by the proxy, and the server-side memory is owned by the remote object. The memory owned by the proxy is freed when the proxy is released by the client program.

A DSOM client can instruct a proxy object to free all memory it owns for the client without releasing the proxy by invoking somdReleaseResources on the proxy object. Calling somdReleaseResources can prevent unused memory from accumulating in a proxy. If a client program repeatedly invokes a remote **get_string** that returns an object-owned string. The proxy will store the memory associated with all of the returned strings until the proxy is released. If the client program only uses the last result returned from **get_string**, then the unused memory accumulates in the proxy. To prevent accumulation invoke **somdReleaseResources** on the proxy object periodically.

Parameters

receiver
A pointer to the **SOMDClientProxy** object to release resources. With C++ bindings, the object is explicitly casted to **SOMDClientProxy,** unless otherwise stated.

ev
A pointer to the **Environment** structure for the method call.

Example

```
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);
```

Original Class

SOMDClientProxy Class

Related Information

release Method

somdTargetFree Method

Forwards the **somFree** method call to the remote target object.

IDL Syntax

void somdTargetFree ( );

Description

somdTargetFree forwards the **somFree** method call to the remote target object, but the proxy object is not destroyed. This method is for when you want to be explicit about freeing the remote target object and not the proxy object.

Parameters

receiver
A pointer to the **SOMDClientProxy** object for the desired remote target object. When invoking this method on an object using the C++ bindings, if the object has not been declared to be of type **SOMDClientProxy** (or some derived type), then it is necessary to explicitly cast the object to **SOMDClientProxy ***.

ev
A pointer to the **Environment** structure for the method caller.

Example

```
#include <somd.h>
#include <car.h>

Environment ev;
Car car;
string somdObjectId;
...
SOM_InitEnvironment (&ev);
SOMD_Init(&ev);
/* restore proxy from its string form */
FileRead(”/u/joe/mycar”, &somdObjectId);
car = _string_to_object(SOMD_ORBObject, &ev, somdObjectId);
...
_somdTargetFree(car, &ev);
```

Original Class

SOMDClientProxy Class

Related Information

release Method

somDestruct Method

SOMFree Function

somdTargetGetClass Method

Returns a proxy for the class object for the remote target object.

IDL Syntax

SOMClass somdTargetGetClass ( );

Description

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 for when the application program wants to be explicit about getting the class object for the remote target object versus the local proxy.

Parameters

receiver
A pointer to the **SOMDClientProxy** object for the desired remote target object. When invoking this method on an object using the C++ bindings, if the object has not been declared to be of type **SOMDClientProxy** (or some derived type), then it is necessary to explicitly cast the object to **SOMDClientProxy ***.

ev
A pointer to the **Environment** structure for the method caller.

Return Value

The **somdTargetGetClass** method returns a proxy to the remote class object for the remote target object. The proxy should be released when the caller is finished using it. (Do not invoke **somFree** or **somDestruct** on the result.)

Example

```
#include <somd.h>
#include <car.h>

Environment ev;
Car car;
SOMClass carClass;
string somdObjectId;
...
SOM_InitEnvironment (&ev);
SOMD_Init(&ev);
/* restore proxy from its string form */
FileRead(”/u/joe/mycar”, &somdObjectId);
car = _string_to_object(SOMD_ORBObject, &ev, somdObjectId);
...
carClass = _somdTargetGetClass(car, &ev);
```

Original Class

SOMDClientProxy Class

Related Information

somdProxyGetClass Method

somdTargetGetClassName Method

Returns the class name for the remote target object.

IDL Syntax

string somdTargetGetClassName ( );

Description

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 versus the proxy object.

Parameters

receiver
A pointer to the **SOMDClientProxy** object for the desired remote target object. When invoking this method on an object using the C++ bindings, if the object has not been declared to be of type **SOMDClientProxy** (or some derived type), then it is necessary to explicitly cast the object to **SOMDClientProxy ***.

ev
A pointer to the **Environment** structure for the method caller.

Return Value

The **somdTargetGetClassName** method returns a string containing the class name of the remote target object. Ownership of this string is given to the caller.

Example

```
#include <somd.h>
#include <car.h>

Environment ev;
Car car;
string carClassName;
string somdObjectId;
...
SOM_InitEnvironment (&ev);
SOMD_Init(&ev);
/* restore proxy from its string form */
FileRead("/u/joe/mycar", &somdObjectId);
car = _string_to_object(SOMD_ORBObject, &ev, somdObjectId);
...
carClassName = _somdTargetGetClassName(car, &ev);
```

Original Class

SOMDClientProxy Class

Related Information

somdProxyGetClassName Method

SOMDObject Class

The SOMDObject class implements the methods that can be applied to all CORBA object references; such as, **duplicate**, **get_implementation**, **get_interface**, **is_nil** and **release**. In the CORBA 1.1 specification, these methods are described in Chapter 8.

In DSOM, there is a subclass of **SOMDObject** called **SOMDClientProxy**. This subclass inherits the implementation of **SOMDObject**, but extends it by overriding **sommProxyDispatch** 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.

File Stem

somdobj

Base

SOMObject Class

Metaclass

SOMClass Class

Ancestor Classes

SOMObject Class

New Methods

create_request Method

create_request_args Method*

duplicate Method

get_implementation Method

get_interface Method

is_nil Method

is_proxy Method*

is_SOM_ref Method*

release Method

(* These methods were added by DSOM to supplement the published CORBA 1.1 interfaces.)

Deprecated Method

is_constant

Overridden Methods

somDefaultInit Method

somDestruct Method

somDumpSelfInt Method

create_request Method

Creates a request to execute a particular operation on the referenced object.

IDL Syntax

ORBStatus create_request (
in Context *ctx*,
in Identifier *operation*,
in NVList *arg_list*,
inout NamedValue *result*,
out Request *request*,
in Flags *req_flags*);

Description

The **create_request** method creates a request to execute a particular operation on the referenced object. Ownership of each input parameter to this method is transferred to the receiver. Hence, the input arguments should not be subsequently freed by the caller.

Parameters

receiver
A pointer to a **SOMObject** object. Even though this method has been introduced by **SOMDObject**, it can be invoked on any **SOMObject** object. When invoking this method on an object using the C++ bindings, if the object has not been declared to be of type **SOMDObject** (or some derived type), then it is necessary to explicitly cast the object to **SOMDObject ***.

ev
A pointer to the **Environment** structure for the method caller. This structure is used only to store exceptions raised during the creation of the Request object, and not exceptions raised subsequently when invoking the Request.

ctx
A pointer to the **Context** object of the requested operation.

operation
The name of the operation to be performed on the target object, *receiver*.

arg_list
A pointer to a list of arguments (**NVList Class**). 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
A pointer to a **NamedValue** structure where the result of applying *operation* to *receiver* should be stored.

request
A pointer to storage for the address of the created **Request** object. Ownership of this parameter is transferred to the caller, which is responsible for destroying it using the **destroy Method** or using **somDestruct Method, somFree Method** or the C++ **delete** operator.

req_flags
A Flags bitmask (unsigned long), which is currently unused.

Return Value

Returns an **ORBStatus** value as the status code for the request.

Example

```
#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 this 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;
SOM_InitEnvironment (&ev);
SOMD_Init(&ev);
/* 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);
/* 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);
```

Original Class

SOMDObject Class

Related Information

create_request_args Method

create_list Method

create_operation_list Method

create_request_args Method

Creates an argument list appropriate for the specified operation.

IDL Syntax

ORBStatus create_request_args (
in Identifier *operation*,
out NVList *arg_list.*
out NamedValue *result*);

Description

The **create_request_args** method creates the appropriate *arg_list* (**NVList Class**) 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.

Parameters

receiver
A pointer to a **SOMObject** object. Even though this method has been introduced by **SOMDObject**, it can be invoked on any **SOMObject** object. When invoking this method on an object using the C++ bindings, if the object has not been declared to be of type **SOMDObject** (or some derived type), then it is necessary to explicitly cast the object to **SOMDObject ***.

ev
A pointer to the **Environment** structure for the method caller.

operation
The Identifier of the operation for which the argument list is being created.

arg_list
A pointer to the location where the method will store a pointer to the resulting argument list. Ownership of this parameter is transferred to the caller, which is responsible for destroying it using the **release Method** or using **somDestruct Method, somFree Method** or the C++ **delete** operator.

result
A pointer to the **NamedValue** structure which will be used to hold the result. The result type field is filled in with the **TypeCode** of the expected result.

Return Value

Returns an **ORBStatus** value representing the return code of the request.

Example

```
#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 this 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;
SOM_InitEnvironment (&ev);
SOMD_Init(&ev);
/* 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);
/* 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);
```

Original Class

SOMDObject Class

Related Information

duplicate Method

create_operation_list Method

create_request Method

release Method

duplicate Method

Makes a duplicate of an object reference.

IDL Syntax

SOMDObject duplicate ( );

Description

The **duplicate** method makes a duplicate of the object reference. Ownership of the returned object is transferred to the caller. The caller should subsequently call the **release** method on this object reference.

Parameters

receiver
A pointer to a **SOMObject** object. Even though this method has been introduced by **SOMDObject**, it can be invoked on any **SOMObject** object. When invoking this method on an object using the C++ bindings, if the object has not been declared to be of type **SOMDObject** (or some derived type), then it is necessary to explicitly cast the object to **(SOMDObject *)**.

ev
A pointer to the **Environment** structure for the method caller.

Return Value

Returns a **SOMDObject** that is a duplicate of the receiver.

If this method is invoked on a proxy object, a new (distinct) proxy object is returned. If the method is invoked on a local **SOMObject**, a pointer to the same object is returned.

Example

```
#include <somd.h>
Environment ev;
SOMObject obj;
SOMDObject objref1, objref2;
...
/* initialization code */
objref1 = _create_SOM_ref(SOMD_SOMOAObject, &ev, obj);
objref2 = _duplicate(objref1,&ev);
...
_release(objref2,&ev);
```

Original Class

SOMDObject Class

Related Information

create Method

create_constant Method

create_SOM_ref Method

release Method

get_implementation Method

Returns the implementation definition for the referenced object.

IDL Syntax

ImplementationDef get_implementation ( );

Description

The **get_implementation** method returns a reference to the implementation definition object for the server in which the object referenced by *receiver* resides. If invoked on an object that is not a proxy or does not reside in a DSOM server process, NULL is returned. When invoked on a proxy object, this method results in a remote invocation and returns a proxy to the remote implementation definition object. Ownership of the returned object is transferred to the caller.

Parameters

receiver
A pointer to a **SOMObject** object. Even though this method has been introduced by **SOMDObject**, it can be invoked on any **SOMObject** object. When invoking this method on an object using the C++ bindings, if the object has not been declared to be of type **SOMDObject** (or some derived type), then it is necessary to explicitly cast the object to **SOMDObject ***.

ev
A pointer to the **Environment** structure for the method caller.

Return Value

Returns a reference to the **ImplementationDef** object for the server in which the remote object referenced by *receiver* resides. NULL is returned if invoked on an object that is not a proxy and does not reside in a DSOM server process.

Example

```
#include <somd.h>

long flags;
Environment ev;
SOMDObject objref;
ImplementationDef impldef;
...
SOM_InitEnvironment (&ev);
SOMD_Init(&ev);
/* code to get objref */
impldef = _get_implementation(objref,&ev);
flags = __get_impl_flags(impldef,&ev);
```

Original Class

SOMDObject Class

Related Information

get_interface Method

get_interface Method

Returns the interface definition object for the referenced object.

IDL Syntax

InterfaceDef get_interface ( );

Description

The **get_interface** method returns the interface definition (**InterfaceDef**) object for the referenced object.

Parameters

receiver
A pointer to a **SOMObject** object. Even though this method has been introduced by **SOMDObject**, it can be invoked on any **SOMObject** object. When invoking this method on an object using the C++ bindings, if the object has not been declared to be of type **SOMDObject** (or some derived type), then it is necessary to explicitly cast the object to **SOMDObject ***.

ev
A pointer to the **Environment** structure for the method caller.

Return Value

The **get_interface** method returns a pointer to the **InterfaceDef** object associated with the reference *receiver,* obtained from the Interface Repository. Ownership of the **InterfaceDef** object is passed to the caller.

If *receiver* is a proxy object, this method results in a remote invocation and returns a proxy to a remote **InterfaceDef** object. Otherwise, if a local Interface Repository is accessible, a local **InterfaceDef** object is returned.

Example

```
#include <somd.h>
#include <repostry.h>
#include <intfacdf.h>

Environment ev;
SOMDObject objref;
InterfaceDef intf;
...
SOM_InitEnvironment (&ev);
SOMD_Init(&ev);
/* code to get objref */
intf = _get_interface(objref,&ev);
```

Original Class

SOMDObject Class

Related Information

get_implementation Method

is_nil Method

Tests to see if the object reference is nil.

IDL Syntax

boolean is_nil ( );

Description

The **is_nil** method tests to see if the specified object reference is nil.

Parameters

receiver
A pointer to any object, either a **SOMObject** or a **SOMDObject**. The pointer can be NULL.

ev
A pointer to the **Environment** structure for the method caller.

Return Value

is_nil returns TRUE if the object reference does not refer to any object (if *receiver* is NULL, OBJECT_NIL, or a **SOMDObject** with no associated server object). Otherwise, **is_nil** returns FALSE.

Example

```
ReferenceData id;
...
/* 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) ||
    _somIsA(objref, SOMDClientProxyNewClass(0, 0)) ||
    _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;
```

Original Class

SOMDObject Class

Related Information

create Method

is_proxy Method

is_SOM_ref Method

is_proxy Method

Tests to see if the object reference is a proxy.

IDL Syntax

boolean is_proxy( );

Description

The **is_proxy** method tests to see if the specified object reference is a proxy object.

Parameters

receiver
A pointer to a **SOMObject** object. Even though this method has been introduced by **SOMDObject**, it can be invoked on any **SOMObject** object. When invoking this method on an object using the C++ bindings, if the object has not been declared to be of type **SOMDObject** (or some derived type), then it is necessary to explicitly cast the object to **SOMDObject ***.

ev
A pointer to the **Environment** structure for the method caller.

Return Value

The **is_proxy** method returns TRUE if the object reference is a proxy object. Otherwise (if *receiver* is a local object or a **SOMDObject** but not a **SOMDClientProxy** object), **is_proxy** returns FALSE.

Example

```
#include <somd.h>

SOMDObject objref;
Environment ev;
Context ctx;
NVlist arglist;
NamedValue result;
Request reqObj;
...
SOM_InitEnvironment (&ev);
SOMD_Init(&ev);
/* code to get objref */
if (_is_proxy(objref, &ev)) {
    /* create a remote request for target object */
...
/* code to create arglist */
   rc = _create_request(obj, &ev, ctx,
                        ”testMethod”, arglist,
                        &result, &reqObj,(Flags)0);
}
```

Original Class

SOMDObject Class

Related Information

is_nil Method

is_SOM_ref Method

string_to_object Method

is_SOM_ref Method

Tests to see if the object reference is a simple reference to a SOM object. This method can only be called from a server.

IDL Syntax

boolean is_SOM_ref ( );

Description

The **is_SOM_ref** method tests to see if the specified object reference is a simple (transient) reference to a SOM object (created via **SOMOA::create_SOM_ref**). This method can only be called from a server; attempting to call **is_SOM_ref** from the client will raise an error.

Parameters

receiver
A pointer to a **SOMDObject** object.

ev
A pointer to the **Environment** structure for the method caller.

Return Value

The **is_SOM_ref** method returns TRUE if the object reference is a simple (transient) reference to a SOM object (created via **SOMOA::create_SOM_ref**). Otherwise, **is_SOM_ref** returns FALSE. An error is raised if **is_SOM_ref** is called from the client.

Example

```
/* inside an override of somdSOMObjFromRef */
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);
```

Original Class

SOMDObject Class

Related Information

create_SOM_ref Method

get_SOM_object Method

is_proxy Method

is_nil Method

release Method

Releases the memory associated with the specified object reference.

IDL Syntax

void release ( );

Description

The **release** method releases the memory associated with the object reference. When invoked on a local object, this method has no effect.

Parameters

receiver
A pointer to a **SOMObject** object. Even though this method has been introduced by **SOMDObject**, it can be invoked on any **SOMObject** object. When invoking this method on an object using the C++ bindings, if the object has not been declared to be of type **SOMDObject** (or some derived type), then it is necessary to explicitly cast the object to **SOMDObject *)**.

ev
A pointer to the **Environment** structure for the method caller.

Example

```
#include <somd.h>

SOMDObject objref;
Environment ev;
SOM_InitEnvironment (&ev);
SOMD_Init(&ev);
...
/* code to get objref */
...
_release(objref, &ev);
```

Original Class

SOMDObject Class

Related Information

duplicate Method

create Method

create_constant Method

create_SOM_ref Method

somdReleaseResources Method

SOMDObjectMgr Class

The **SOMDObjectMgr** class is derived from **ObjectMgr** class and provides the DSOM implementations for the **ObjectMgr** methods. This class and all of its methods have been deprecated. Use of the methods is discouraged, but the methods are supported in the current release of SOMobjects.

File Stem

somdom

Base

ObjectMgr Class

Metaclass

SOMMSingleInstance Metaclass

Ancestor Classes

ObjectMgr Class

SOMObject Class

Attribute

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 Method**, when invoked on a proxy object, will free the proxy object along with the remote object. The default value is TRUE, indicating that both the remote object and the proxy object are freed. Setting this attribute to FALSE as part of client-program initialization, for example:

```
__set_somd21somdFree(SOMD_ObjectMgr, ev, FALSE);
```

restores the DSOM 2.x default semantics in which the remote object is freed but the proxy is not.

Deprecated Methods

Use of the **SOMDObjectMgr** methods listed below is discouraged.

somdFindAnyServerByClass*

somdFindServer

somdFindServersByClass

somdFindServerByName*

Overridden Methods

somDefaultInit Method

SOMDServer Class

The **SOMDServer** class is a base class that defines and implements methods for managing objects in a DSOM server process. This includes methods for mapping between object references and SOM objects, dispatching methods on objects, and creating factories in servers.

This class should be subclassed in order to customize the creation of object references and to facilitate object activation, method dispatching, and factory creation. The **ImplementationDef** object of a server indicates which type of server object (which subclass of **SOMDServer**) the server will use (via the **impl_server_class** attribute)**.** After a server program has invoked **impl_is_ready**, the **SOMD_ServerObject** global variable refers to the server's server object.

File Stem

somdserv

Base

SOMObject Class

Metaclass

SOMMSingleInstance Metaclass

Ancestor Classes

SOMObject Class

New Methods

somdCreateFactory Method*

somdDispatchMethod Method*

somdObjReferencesCached Method*

somdRefFromSOMObj Method*

somdSOMObjFromRef Method*

(*This class and its methods were added by DSOM to supplement the published CORBA 1.1 interfaces.)

Deprecated Methods

Use of the **SOMDServer** methods listed below is discouraged.

somdCreateObj

somdDeleteObj

somdGetClassObj

somdCreateFactory Method

Creates a factory object that can create objects of the specified class.

IDL Syntax

SOMObject somdCreateFactory (
in string *className*,
in ExtendedNaming::PropertyList *props*);

Description

The **somdCreateFactory** method is called by DSOM to dynamically create a factory in the server. Two kinds of factories can be created and returned by the default implementation of **somdCreateFactory**: an application-specific factory or a SOM class object. The default implementation uses the IDL modifier **factory** to map from the given class name to an application-specific factory class name. If this modifier is not specified, the SOM class object is returned as the default factory.

This method is not intended to be called directly by applications. Rather, it is called by DSOM and is intended to be overridden in user-defined subclasses of **SOMDServer** to customize the way factories are created in a specific server.

Parameters

receiver
A pointer to an object of class **SOMDServer**.

ev
A pointer to the **Environment** structure for the caller.

className
The name of the class for which a factory is needed.

props
A complete list of the Naming Service properties registered in the DSOM Factory Service for the requested class.

Return Value

The **somdCreateFactory** method returns a factory object for the specified class.

Example

```
#include <somd.h>

/* Override somdCreateFactory to customize the way in which
 * factories are associated with classes or the way factories
 * are created.*/

SOM_Scope SOMObject SOMLINK somdCreateFactory(myServer somSelf,
              Environment *ev, string className,
              ExtendedNaming_PropertyList *props)
{
    somId factoryId;
    SOMObject factory;

    /* override factory modifier for Animal class */
    if (strcmp(className, "Animal") == 0) {
       factoryId = somIdFromString("AnimalFactory");
       factory = _somFindClass(SOMClassObjectMgr, classId, 0, 0);
       SOMFree(factoryId);
       if (factory)
```

```
                return _somNew(factory);
            else
                return NULL;
         }
         else {
            return  myServer_parent_SOMDServer_somdCreateFactory(
                                    somSelf, ev, className, props);
         }
     }
```

Original Class

SOMDServer Class

Related Information

somdCreate Function

somdDispatchMethod Method

Dispatches a method on the specified SOM object.

IDL Syntax

void somdDispatchMethod (
in SOMObject ***somobj,***
out somToken ***retValue,***
in somId ***methodId,***
in va_list ***ap*** **);**

Description

This method is not intended to be called directly. (It is called by the DSOM run time.) Instead, it is intended to be overridden by user-defined classes of **SOMDServer** to customize method dispatching in a server.

The **somdDispatchMethod** method intercepts method calls on objects in a server. When a request arrives in a server, DSOM extracts the request parameters from the message, and resolves the target object. Then, DSOM dispatches the method call on the target object by calling the **somdDispatchMethod** method.

The default implementation calls **somDispatch** on the target object with the parameters as specified. This method can be overridden to intercept and process the method calls before or after they are dispatched.

Parameters

receiver
A pointer to a **SOMDServer** object.

ev
A pointer to the **Environment** structure for the method caller.

somobj
A pointer to an object “managed” by the server object.

retValue
A pointer to the storage area allocated to hold the method result value, if any.

methodId
A **somId** for the name of the method which is to be dispatched.

ap
A pointer to a **va_list** array of arguments to the method call.

Return Value

The **somdDispatchMethod** method will return a result, if any, in the storage whose address is in *retValue.*

Example

This is an example of overriding **SOMDispatchMethod** to print a message in the server before and after dispatching every method.

```
#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",
       somStringFromId(methodId), somobj);
SOMObject_somDispatch(somobj, ev, retValue, methodId, ap);
printf("finished dispatching %s on %x\n",
       somStringFromId(methodId, somobj);
}
```

Original Class

SOMDServer Class

somdObjReferencesCached Method

Indicates whether a server object retains ownership of the object references it creates via the **somdRefFromSOMObj** method.

IDL Syntax

boolean somdObjReferencesCached ( );

Description

The **somdObjReferencesCached** 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.

Parameters

receiver
A pointer to an object of class **SOMDServer.**

ev
A pointer to the **Environment** structure for the calling method.

Return Value

The method returns FALSE by default; overriding implementations may return TRUE to indicate that a subclass of **SOMDServer** implements object reference caching.

Example

```
SOMDobject objref;
objref = _somdRefFromSOMObj(SOMD_ServerObject, ev, myobj);
...
/* code to use objref */
...
if (!_somdObjReferencesCached(SOMD_ServerObject, ev))
    _release(objref, ev);
```

Original Class

SOMDServer Class

Related Information

somdRefFromSOMObj Method

somdRefFromSOMObj Method

Returns an object reference corresponding to the specified SOM object.

IDL Syntax

SOMDObject somdRefFromSOMObj (
in SOMObject *somobj*);

Description

The **somdRefFromSOMObj** method creates a reference to a SOM object in a server, to be exported to a client as a proxy. This method is called by DSOM as part of converting the results of a local method call into a result message for a remote client, whenever the result contains a pointer to an object local to the server. The default implementation creates simple (transient) references. This method is intended to be overridden in user-defined subclasses of **SOMDServer** to customize the creation of object references.

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.

Parameters

receiver
A pointer to a **SOMDServer** object.

ev
A pointer to the **Environment** structure for the method caller.

somobj
A pointer to the SOM object for which a DSOM reference is to be created.

Return Value

The **somdRefFromSOMObj** method returns a DSOM reference for the SOM object specified.

Example

```
SOM_Scope SOMDObject SOMLINK
              somdRefFromSOMObj(SOMPServer somSelf,
                                Environment *ev,
                                SOMObject obj)
{
  SOMDObject objref;
  Repository repo;
  repo = SOM_InterfaceRepository;
/* is obj persistent */
  if (object_is_persistent(obj, ev)) {
    /* Create an object reference based on persistent ID. */
    ReferenceData rd = create_refdata_from_object(ev, obj);

    InterfaceDef intf =
       _lookup_id(repo,ev,
                  somGetClassName(obj));
    objref = _create (SOMD_SOMOAObject, ev, &rd,
                            intf, SOMD_ImplDefObject);
    _somFree(intf);
    _somFree(repo);
    SOMFree(rd._buffer);
```

```
  } else /* obj is not persistent, so get Ref in usual way */
     objref = parent_somdRefFromSOMObj(somSelf, ev, obj);
  return(objref);
}
```

Original Class

SOMDServer Class

Related Information

somdSOMObjFromRef Method

somdObjReferencesCached Method

create Method

create_constant Method

create_SOM_ref Method

somdSOMObjFromRef Method

Returns the SOM object corresponding to the specified object reference.

IDL Syntax

SOMObject somdSOMObjFromRef (
in SOMDObject *objref*);

Description

The **somdSOMObjFromRef** method returns the SOM object associated with the DSOM object reference, *objref*. This method is called by **SOMOA Class** as part of converting a remote request into a local method call on an object in a server. This method is intended to be overridden in user-defined subclasses of **SOMDServer** to customize the creation and resolution of object references.

Parameters

receiver
A pointer to a **SOMDServer** object.

ev
A pointer to the **Environment** structure for the method caller.

objref
pointer to the DSOM object reference to the SOM object.

Return Value

The **somdSOMObjFromRef** method returns the SOM object associated with the supplied DSOM reference.

Example

```
SOM_Scope SOMObject SOMLINK
              somdSOMObjFromRef(SOMPServer somSelf,
                                Environment *ev,
                                SOMDObject objref)
{  SOMObject obj;
        if (_is_nil(objref, ev))
      return (SOMObject *) NULL;
   /* Make sure this isn't a local object or proxy: */
   if (!_somIsA(objref, _SOMDObject) || _is_proxy(objref, ev))
      return objref;

   /* test if objref is mine */
   if (!_is_SOM_ref(objref, ev)) {
      /* objref was mine, activate persistent object myself */
      ReferenceData rd = _get_id(SOMD_SOMOAObject, ev, objref);
      obj = get_object_from_refdata(ev, &rd);
      SOMFree(rd._buffer);
   } else
     /* it's not one of mine, let parent activate object */
     obj = parent_somdSOMObjFromRef(somSelf, ev, objref);
   return obj;
}
```

Original Class

SOMDServer Class

Related Methods

somdRefFromSOMObj Method

get_id Method

is_SOM_ref Method

SOMDServerMgr Class

The **SOMDServerMgr** class provides a programmatic interface to manage server processes. The server processes that can be managed are limited to those registered in the DSOM Factory Service through the **regimpl** or the **ImplRepository** interface.

File Stem

servmgr

Base

SOMObject Class

Metaclass

SOMClass Class

Ancestor Classes

SOMObject Class

New Methods

somdListServer Method

somdRestartServer Method

somdShutdownServer Method

somdStartServer Method

Deprecated Methods

somdDisableServer

somdEnableServer

somdIsServerEnabled

somdListServer Method

Queries the state of a server process.

IDL Syntax

ORBStatus somdListServer (in string *server_alias*);

Description

The **somdListServer** method queries the status of the server process associated with the server alias. If multiple servers with the same alias are registered in the Factory Service, the status of the first one found is queried. If the server process is running, the return code is 0 indicating success.

Parameters

receiver
A pointer to an object of class **SOMDServerMgr.**

ev
A pointer to the **Environment** structure for the calling method.

server_alias
The implementation alias of the server to be listed.

Return Value

Returns 0 if the server process is running; otherwise, a DSOM error code is returned.

Example

```
#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);
```

Original Class

SOMDServerMgr Class

somdRestartServer Method

Restarts a server process. The **somdRestartServer** method should only be called when the server has finished servicing all outstanding requests.

IDL Syntax

ORBStatus somdRestartServer (in string *server_alias*);

Description

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.

If the server is not responding to requests (hung), this method will fail to restart the server and will return an error after the request has timed out.

If multiple servers with the given *server_alias* are registered in the Factory Service, this method attempts to restart the first one found.

Parameters

receiver
A pointer to an object of class **SOMDServerMgr.**

ev
A pointer to the **Environment** structure for the calling method.

server_alias
The implementation alias of the server to be restarted.

Return Value

Returns 0 for success or a DSOM error code for failure.

Example

```
#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);
```

Original Class

SOMDServerMgr Class

somdShutdownServer Method

Stops a server process.

IDL Syntax

ORBStatus somdShutdownServer (in string *server_alias*);

Description

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.

If the server is not responding to requests (hung), this method will fail to stop the server and will return an error after the request has timed out.

If multiple servers with the given *server_alias* are registered in the Factory Service, this method attempts to stop the first one found.

Parameters

receiver
A pointer to an object of class **SOMDServerMgr.**

ev
A pointer to the **Environment** structure for the calling method.

server_alias
The implementation alias of the server to be stopped.

Return Value

Returns 0 for success or a DSOM error code for failure.

Example

```
#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);
```

Original Class

SOMDServerMgr Class

somdStartServer Method

Starts a server process.

IDL Syntax

ORBStatus somdStartServer (in string *server_alias*);

Description

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.

If multiple servers with the given *server_alias* are registered in the Factory Service, this method attempts to start the first one found.

Parameters

receiver
A pointer to an object of class **SOMDServerMgr.**

ev
A pointer to the **Environment** structure for the calling method.

server_alias
The implementation alias of the server to be started.

Return Value

Returns 0 for success or a DSOM error code for failure.

Example

```
#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);
```

Original Class

SOMDServerMgr Class

SOMOA Class

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.

Some methods of **SOMOA** are intended to be called directly from user-written server programs (for example, **activate_impl_failed, execute_next_request**, and **execute_request_loop**) while others are intended to be called from user-written subclasses of **SOMDServer, create_constant, create_SOM_ref**, and **get_SOM_object**).

File Stem

somoa

Base

BOA Class

Metaclass

SOMMSingleInstance Metaclass

Ancestor Classes

BOA Class

SOMObject Class

New Methods

activate_impl_failed Method*

create_constant Method*

create_SOM_ref Method*

execute_next_request Method*

execute_request_loop Method*

get_SOM_object Method*

(* This class and its methods were added by DSOM to supplement the published CORBA 1.1 interfaces.)

Overridden Methods

change_implementation Method

create Method

deactivate_impl Method

deactivate_obj Method

dispose Method

get_id Method

get_principal Method

impl_is_ready Method

obj_is_ready Method

obj_is_ready Method

somDefaultInit Method

somDestruct Method

Deprecated Methods

change_id Method

create_constant

change_id Method

activate_impl_failed Method

Sends a message to the DSOM daemon indicating that a server did not activate.

IDL Syntax

void activate_impl_failed (
in ImplementationDef *implDef*,
in long *rc*);

Description

The **activate_impl_failed** method sends a message to the DSOM daemon (**somdd**) indicating that the server did not activate. This method should be called from a server program if the server terminates prior to calling **impl_is_ready**. After **impl_is_ready** has been successfully invoked, servers should invoke **deactivate_impl** rather than **activate_impl_failed** upon termination.

Parameters

receiver
A pointer to the **SOMOA** object that attempted to activate the implementation.

ev
A pointer to the **Environment** structure for the method caller.

implDef
A pointer to the **ImplementationDef** object representing the implementation (server) that failed to activate.

rc
A return code designating the reason for failure.

Example

```
#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);
```

Original Class

SOMOA Class

create_SOM_ref Method

Creates a simple, transient DSOM reference to a SOM object in a server.

IDL Syntax

SOMDObject create_SOM_ref (
in SOMObject *somobj,*
in ImplementationDef *impl*);

Description

The **create_SOM_ref** method creates a simple DSOM reference for a local SOM object in a server. There is no user-defined **ReferenceData** associated with the object, and this object reference is only valid while the target SOM object and the server are active.

The **SOMObject** associated with the **SOM_ref** can be retrieved by using the **get_SOM_object** method. The **is_SOM_ref** method of **SOMDObject** can be used to determine whether the reference was created using **create_SOM_ref** or not.

Ownership of the new object reference is transferred to the caller.

Parameters

receiver
A pointer to the **SOMOA** object managing the implementation.

ev
A pointer to the **Environment** structure for the method caller.

somobj
A pointer to the local **SOMObject** to be referenced.

impl
A pointer to the **ImplementationDef** of the calling server process.

Return Value

The **create_SOM_ref** method returns a pointer to a **SOMDObject**.

Example

```
SOMDObject objref;
...
/* you 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);
```

Original Class

SOMOA Class

Related Information

create Method

create_constant Method

get_SOM_object Method

is_SOM_ref Method

execute_next_request Method

Instructs DSOM to receive and execute the next client request.

IDL Syntax

ORBStatus execute_next_request (in Flags *waitFlag*);

Description

The **execute_next_request** method receives the next request message, executes the request, and sends the result to the caller. This method is intended to be called by a server program, to instruct DSOM to receive and dispatch the next client request.

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.

Parameters

receiver
A pointer to the **SOMOA** object managing the implementation.

ev
A pointer to the **Environment** structure for the method caller.

waitFlag
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).

Return Value

The **execute_next_request** method 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.

Example

```
#include <somd.h>

/* server initialization code ... */
SOM_InitEnvironment(&ev);
SOMD_Init (&ev) ;
/* code to initialize SOMD_ImplDefObject */
/* 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++;
}
```

Original Class

SOMOA Class

Related Information

execute_request_loop Method

execute_request_loop Method

Instructs DSOM to repeatedly receive and service client requests.

IDL Syntax

ORBStatus execute_request_loop (in Flags *waitFlag*);

Description

execute_request_loop initiates a loop that repeatedly waits for a request message, executes the request and then returns the result to the client. Server program use this method to instruct DSOM to receive and dispatch the next client request.

When called with the SOMD_WAIT flag, this method loops infinitely (or until an error or until the server is terminated).

When called with the SOMD_NO_WAIT flag, this method loops as long as it finds a request message to process. This is useful when writing event-driven applications where there are event sources other than DSOM requests, for 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 and then return control to the main application.

If the server's **ImplementationDef** indicates the server is multithreaded each request will be run by **SOMOA** in a separate thread.

Parameters

receiver
A pointer to the **SOMOA** object managing the implementation.

ev
A pointer to the **Environment** structure for the method caller.

waitFlag
A Flags bitmask indicating if the method should block (SOMD_WAIT) or return to the caller (SOMD_NO_WAIT) when there is no request message pending.

Return Value

Returns an OBJ_ADAPTER exception that contains a DSOM error code for the operation. If the invocation uses SOMD_NO_WAIT and no message is pending, an **ORBStatus** code SOMDERROR_NoMessages is returned.

Example

```
#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);
```

Original Class

SOMOA Class

Related Information

SOMD_RegisterCallback Function

execute_next_request Method

get_SOM_object Method

Gets the SOM object in a server associated with a simple DSOM reference.

IDL Syntax

SOMObject get_SOM_object (in SOMDObject *somref*);

Description

The **get_SOM_object** method returns the SOM object associated with a reference created by the **create_SOM_ref** method. This method should not be invoked on object references created using **create** or **create_constant**. This method can be used in subclasses of **SOMDServer**, in the implementation of the **somdSOMObjFromRef** method, to determine whether or not to make a parent-method call.

Parameters

receiver
A pointer to the **SOMOA** object managing the implementation.

ev
A pointer to the **Environment** structure for the method caller.

somref
A pointer to a **SOMDObject** created by the **create_SOM_ref** method.

Return Value

The **get_SOM_object** method returns the SOM object associated with the reference.

Example

```
/* Within somdSOMObjFromRef:  */

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);
...
```

Original Class

SOMOA Class

Related Information

create_SOM_ref Method

is_SOM_ref Method

Chapter 3. Interface Repository Framework Classes

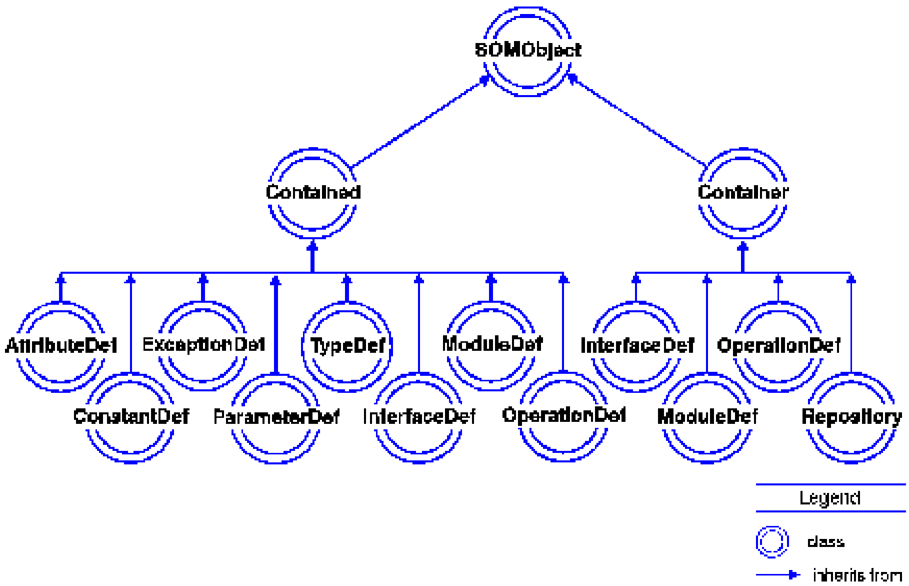

Figure 3. Interface Repository Framework Class Organization.

AttributeDef Class

The **AttributeDef** class provides the interface for attribute definitions.

File Stem

attribdf

Base

Contained Class

Metaclass

SOMClass Class

Ancestor Classes

Contained Class

SOMObject Class

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.

Attributes

Listed below is each available attribute, with its corresponding type in parentheses, followed by a description of its purpose:

type (TypeCode)
The **TypeCode** of the attribute. The **TypeCode** returned by the **_get_** form of the type attribute is contained in the receiving **AttributeDef** object that retains ownership. The returned TypeCode should not be freed. To obtain a 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).

Overriding Methods

somDefaultInit Method

somDestruct Method

somDumpSelf Method

somDumpSelfInt Method

describe Method

ConstantDef Class

The **ConstantDef** class provides the interface for constant definitions in the Interface Repository.

File Stem

constdef

Base

Contained Class

Metaclass

SOMClass Class

Ancestor Classes

Contained Class

SOMObject Class

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.

Attributes

Listed below is each available attribute, with its corresponding type in parentheses, followed by a description of its purpose:

type (TypeCode)
The **TypeCode** of constant. The **TypeCode** returned by the **_get_** form of the type attribute is contained in the receiving ConstantDef object that retains ownership. 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.

Overriding Methods

somDefaultInit Method

somDestruct Method

somDumpSelf Method

somDumpSelfInt Method

describe Method

Contained Class

The **Contained** class is the most generic form of interface for objects in SOM's CORBA-compliant Interface Repository (IR). All objects contained in the IR inherit this interface.

File Stem

containd

Base

SOMObject Class

Metaclass

SOMClass Class

Ancestor Classes

SOMObject Class

Types

```
typedef  string  RepositoryId;
struct  Description   {
     Identifier  name;
     any    value;
};
```

Attributes

All attributes of the 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 Method**). 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 Class** name or **InterfaceDef Class** 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 RepositoryIds 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 Class** 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. This attribute is a SOM-unique extension of the Interface Repository; it is not stipulated by the CORBA specification.

New Methods

within Method

describe Method

Overriding Methods

somDefaultInit Method

somDestruct Method

somDumpSelf Method

somDumpSelfInt Method

somFree Method

describe Method

Returns a structure containing information defined in the IDL specification that corresponds to a specified **Contained** object in the Interface Repository.

IDL Syntax

Description describe ( );

Description

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);
```

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.

Parameters

receiver
A pointer to the **Contained** object in the Interface Repository for which a **Description** is needed**.**

ev
A pointer to the **Environment** structure for the caller.

Return Value

The **describe** method returns a structure of type **Description** containing information defined in the IDL specification of the receiving object.

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 Class,** 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.

Example

Here is a code fragment written in C that uses the **describe** method:

```
#include <containd.h>
#include <attribdf.h>
#include <somtc.h>

. . .
AttributeDef attr;  /* An AttributeDef object (also 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 */
```

Original Class

Contained Class

Related Information

within Method

within Method

Returns a list of objects (in the Interface Repository) that contain a specified **Contained** object.

IDL Syntax

sequence<Container> within ( );

Description

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 Class** or **ModuleDef Class**, 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 **Containers** 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++)
         _somFree (seq._buffer[i]); /* Release each
                                       Container obj   */
    SOMFree (seq._buffer);        /* Release the sequence buffer */
}
```

Parameters

receiver
A pointer to a **Contained** object for which containing objects are needed.

ev
A pointer to the **Environment** structure for the caller.

Return Value

The **within** method returns a sequence of **Container** objects that contain the specified **Contained** object.

Example

Here is a code fragment written in C that uses the **within** method:

```
#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<sc._length; i++) {
     printf ("\t%s\n",
         Contained__get_name ((Contained) sc._buffer[i], ev));
     SOMObject_somFree (sc._buffer[i]);
}
```

```
if (sc._length)
     SOMFree (sc._buffer);
```

Original Class

Contained Class

Related Information

describe Method

Container Class

The **Container** class is a generic interface that is common to all of the SOM CORBA-compliant Interface Repository (IR) objects that can hold or contain other objects. A **Container** object can be one of three types: **ModuleDef Class, InterfaceDef Class** or **OperationDef Class.**

File Stem

containr

Base

SOMObject Class

Metaclass

SOMClass Class

Ancestor Classes

SOMObject Class

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

contents Method

lookup_name Method

describe_contents Method

Overriding Methods

somDefaultInit Method

somDestruct Method

somDumpSelf Method

somDumpSelfInt Method

contents Method

Returns a sequence indicating the objects contained within a specified **Container** object of the Interface Repository.

IDL Syntax

sequence<Contained> contents (
in InterfaceName *limit_type*,
in boolean *exclude_inherited*);

Description

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 Class** object, this method can list all of the objects in the Repository, then all of the objects within the **ModuleDef Class** objects, then all within the **InterfaceDef Class** objects, and so on.

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  */
}
```

Parameters

receiver
A pointer to a **Container** object whose contained objects are needed**.**

ev
A pointer to the **Environment** structure for the caller.

limit_type
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
A boolean value: TRUE to exclude any inherited objects, or FALSE to include all objects.

Return Value

The **contents** method returns a sequence of pointers to objects contained within the specified **Container** object.

Example

Here is a code fragment written in C that uses the **contents** method:

```
#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<sc._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”);
```

Original Class

Container Class

Related Information

lookup_name Method

describe_contents Method

describe_contents Method

Returns a sequence of descriptions of the objects contained within a specified **Container** object of the Interface Repository.

IDL Syntax

sequence<ContainerDescription> describe_contents (
in InterfaceName *limit_type*,
in boolean *exclude_inherited,*
in long *max_returned_objs*);

Description

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  */
}
```

Parameters

receiver
A pointer to a **Container** object whose contained object descriptions are needed**.**

ev
A pointer to the **Environment** structure for the caller.

limit_type
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
A boolean value: TRUE to exclude any inherited objects, or FALSE to include all objects.

max_returned_objs
A long integer indicating the maximum number of objects to be returned by the method, or -1 to indicate no limit is set.

Return Value

The **describe_contents** method 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.

Example

Here is a code fragment written in C that uses the **describe_contents** method:

```
#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<sc._length; 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");
```

Original Class

Container Class

Related Information

contents Method

describe Method

lookup_name Method

lookup_name Method

Locates an object by name within a specified **Container** object of the Interface Repository, or within objects contained in the **Container** object.

IDL Syntax

sequence<Contained> lookup_name (
in Identifier *search_name,*
in long *levels_to_search*,
in InterfaceName *limit_type*,
in boolean *exclude_inherited*);

Description

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  */
    }
```

Parameters

receiver
A pointer to a **Container** object in which to locate the object.

ev
A pointer to the **Environment** structure for the caller.

search_name
The name of the object to be located.

levels_to_search
A long having the value 1 or -1.

limit_type
The name of one interface type (see the valid list above) or all, to specify what type of object to search for.

exclude_inherited
A boolean value: TRUE to exclude an object when it is inherited, or FALSE to return the object from wherever it is found.

Return Value

The **lookup_name** method 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.

Example

Here is a code fragment written in C that uses the lookup_name method:

```
#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<sc._length; i++) {
    printf ("\t%s\n",
        Contained__get_id (sc._buffer[i], ev));
    SOMObject_somFree (sc._buffer[i]);
}
if (sc._length)
    SOMFree (sc._buffer);
```

Original Class

Container Class

Related Information

contents Method

describe_contents Method

irindex

ExceptionDef Class

The **ExceptionDef** class provides the interface for exception definitions in the Interface Repository.

File Stem

excptdef

Base

Contained Class

Metaclass

SOMClass Class

Ancestor Classes

Contained Class

SOMObject Class

Types

```
struct ExceptionDescription   {
     Identifier name;
     RepositoryIdid;
     RepositoryIddefined_in;
     TypeCodetype;
};
```

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 Function** 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**.

Overriding Methods

somDefaultInit Method

somDestruct Method

somDumpSelf Method

somDumpSelfInt Method

describe Method

InterfaceDef Class

The **InterfaceDef** class provides the interface for interface definitions in the Interface Repository.

File Stem

intfacdf

Base

Contained Class

Container Class

Metaclass

SOMClass Class

Ancestor Classes

Contained Class

Container Class

SOMObject Class

Types

```
struct FullInterfaceDescription   {
     Identifier name;
     RepositoryIdid;
     RepositoryIddefined_in;
     sequence<OperationDef::OperationDescription>  operation;
     sequence<AttributeDef::AttributeDescription>  attributes;
};
struct InterfaceDescription   {
     Identifier name;
     RepositoryIdid;
     RepositoryIddefined_in;
};
```

The **describe** method, inherited from **Contained,** returns an **InterfaceDescription** structure in the “value” member of the **Description** structure.

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 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 Method**). 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 RepositoryIds 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. This attribute is a SOM-unique extension of the Interface Repository; it is not stipulated by the CORBA specifications.

New Methods

describe_interface Method

Overriding Methods

somDefaultInit Method

somDestruct Method

somDumpSelf Method

somDumpSelfInt Method

within Method

describe Method

describe_contents Method

describe_interface Method

Returns a description of all the methods and attributes of an interface definition.

IDL Syntax

FullInterfaceDescription describe_interface ( );

Description

describe_interface 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 by:

```
if (fid.operation._length)
    SOMFree (fid.operation._buffer);      /* Release the buffer */
if (fid.attributes._length)
    SOMFree (fid.attributes._buffer);     /* Release the buffer */
```

Parameters

receiver
A pointer to an object of class **InterfaceDef** representing the Interface Repository object where an interface definition is stored.

ev
A pointer where the method can return exception information if an error is encountered.

Return Value

A description of all the methods and attributes of an interface definiitions held in the IR.

Example

A C code fragment that uses **describe_interface**:

```
#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.operation._length; i++)
        printf ("\t%s\n", fid.operation._buffer[i].name);
    SOMFree (fid.operation._buffer);
}
```

Original Class

InterfaceDef Class

ModuleDef Class

The **ModuleDef** class provides the interface for module definitions in the Interface Repository.

File Stem

moduledf

Base

Contained Class

Container Class

Metaclass

SOMClass Class

Ancestor Classes

Contained Class

Container Class

SOMObject Class

Types

```
struct ModuleDescription   {
     Identifier name;
     RepositoryIdid;
     RepositoryIddefined_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.

Overriding Methods

somDefaultInit Method

somDestruct Method

somDumpSelf Method

somDumpSelfInt Method

describe Method

describe_contents Method

within Method

OperationDef Class

The **OperationDef** class provides the interface for operation (method) definitions in the Interface Repository.

File Stem

operatdf

Base

Contained Class

Container Class

Metaclass

SOMClass Class

Ancestor Classes

Contained Class

Container Class

SOMObject Class

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** of the operation (method).The **TypeCode** returned by the **_get_** form of the type attribute is contained in the receiving **OperationDef** object that retains ownership. The returned **TypeCode** should not be freed. To obtain a copy, use the **TypeCode_copy Function** 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 ContextIdentifiers 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.

Overriding Methods

somDefaultInit Method

somDestruct Method

somDumpSelf Method

somDumpSelfInt Method

describe Method

describe_contents Method

ParameterDef Class

The **ParameterDef** class provides the interface for parameter definitions.

File Stem

paramdef

Base

Contained Class

Metaclass

SOMClass Class

Ancestor Classes

Contained Class

SOMObject Class

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.

Attributes

Listed below is each available attribute, with its corresponding type in parentheses, followed by a description of its purpose:

type (TypeCode)
The **TypeCode** of the parameter. The **TypeCode** returned by the **_get_** form of the type attribute is contained in the receiving **ParameterDef** object that retains ownership. You should not free the returned **TypeCode**. To obtain a copy, use the **TypeCode_copy Function** 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).

Overriding Methods

somDefaultInit Method

somDestruct Method

somDumpSelf Method

somDumpSelfInt Method

describe Method

Repository Class

The **Repository** class provides global access to SOM's CORBA-compliant Interface Repository (IR) discussed in "The Interface Repository Framework" on page 339 of *Programmer's Guide for SOM and DSOM.*

File Stem

repostry

Base

Container Class

Metaclass

SOMClass Class

Ancestor Classes

Container Class

SOMObject Class

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

lookup_id Method

lookup_modifier Method

release_cache Method

Overriding Methods

describe_contents Method

somDefaultInit Method

somDestruct Method

somFree Method

somDumpSelf Method

somDumpSelfInt Method

lookup_id Method

Returns the object having a specified **RepositoryId**.

IDL Syntax

Contained lookup_id (in RepositoryId *search_id);*

Description

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**.

Parameters

receiver
A pointer to an object of class **Repository** representing SOM's Interface Repository**.**

ev
A pointer where the method can return exception information if an error is encountered.

search_id
An ID value of type **RepositoryId** that uniquely identifies the desired object in the Interface Repository.

Return Value

The **lookup_id** method returns the **Contained Class** object that has the specified **RepositoryId**.

Example

Here is a code fragment written in C that uses the **lookup_id** method:

```
#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);
}
```

Original Class

Repository Class

Related Information

lookup_modifier Method

lookup_name Method

contents Method

within Method

lookup_modifier Method

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].

IDL Syntax

string lookup_modifier (
in RepositoryId *id*,
in string *modifier*);

Description

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 “Modifier Statements” on page 133 in *Programmer’s Guide for SOM and DSOM.*

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.

The **lookup_modifier** method is not stipulated by the CORBA specifications; it is a SOM-unique extension to the Interface Repository.

Parameters

receiver
A pointer to an object of class **Repository** representing SOM’s Interface Repository.

ev
A pointer where the method can return exception information if an error is encountered.

id
The **RepositoryId** of the object whose modifier value is needed.

modifier
The name of a specific (SOM or user-specified) modifier whose string value is needed.

Return Value

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 Method**).

Example

Here is a code fragment written in C that uses the **lookup_modifier** method:

```
#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);
```

Original Class

Repository Class

Related Information

lookup_id Method

lookup_name Method

release_cache Method

Permits the Repository object to release the memory occupied by Interface Repository objects that have been implicitly referenced.

IDL Syntax

void release_cache ( );

Description

This method allows the Repository object to release the memory occupied by implicitly referenced Interface Repository objects. Some methods (such as **describe_contents Method** and **lookup_name Method**) 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.

Parameters

receiver
A pointer to an object of class **Repository** representing SOM's Interface Repository.

ev
A pointer where the method can return exception information if an error is encountered.

Example

```
#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);
```

Original Class

Repository Class

Related Information

See "A Word about Memory Management" on page 348 in *Programmer's Guide for SOM and DSOM.*

TypeDef Class

The **TypeDef** class provides the interface for **typedef** definitions in the Interface Repository. The *TypeCode_xxx* function definitions are located in the `somtc.h` and `somtc.xh` header files.

File Stem

typedef

Base

Contained Class

Metaclass

SOMClass Class

Ancestor Classes

Contained Class

SOMObject Class

Types

```
struct TypeDescription   {
     Identifier name;
     RepositoryIdid;
     RepositoryIddefined_in;
     TypeCodetype;
};
```

The **describe** method, inherited from **Contained,** returns a **TypeDescription** structure in the value member of the **Description** structure.

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 Function** 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**.

Overriding Methods

somDefaultInit Method

somDestruct Method

somDumpSelf Method

somDumpSelfInt Method

describe Method

TypeCodeNew Function

Creates a new **TypeCode** instance.

C Syntax

TypeCode TypeCodeNew (TCKind *tag*, ...);

[The actual parameters indicated by "..." are variable in number and type, depending on the value of the *tag* parameter.] There are no implicit parameters to this function.

TypeCodeNew (tk_objref, string *interfaceId*);
TypeCodeNew (tk_string, long *maxLength*);
TypeCodeNew (tk_sequence, TypeCode *seqTC*, long *maxLength*);
TypeCodeNew (tk_array, TypeCode *arrayTC*, 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*, long *flag*,
long *labelValue*, string *mbrName*, TypeCode *mbrTC*, [...,]
[*flag, labelValue, mbrName* and *mbrTC* repeat as needed]
NULL);
TypeCodeNew (tk_enum, string *name*,
string *enumId*, [...,]
[*enumId*s repeat as needed]
NULL);
TypeCodeNew (TCKind *allOtherTagValues*);

Description

The **TypeCodeNew** function creates a new instance of a **TypeCode** from the supplied parameters. **TypeCodes** 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. There are no implicit parameters to this function. All **TypeCode**s created by **TypeCodeNew** should be destroyed using **TypeCode_free**.

This function is a SOM-unique extension to the CORBA standard.

Parameters

tag
The type or category of **TypeCode** to create.

interfaceId
A string containing the fully-qualified interface name that is the subject of an object reference type.

name
A string that gives the name of a **struct, union** or **enum**.

mbrName
A string that gives the name of a **struct** or **union** member element.

enumId
A string that gives the name of an enum enumerator.

structOrUnionName
A string that gives the name of a struct or union that has been previously named in the current **TypeCode** and is the subject of a self-referential pointer type. See **tk_self** in the table given in the **TypeCode_kind** function description for an example of what this means and how it is applied.

maxLength
The maximum permitted length of a string or a sequence. The value 0 (zero) means the string or sequence is considered unbounded.

length
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
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
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
A **TypeCode** representing the data type of a struct or union member.

swTC
A **TypeCode** that represents the data type of the discriminator in an IDL **union** statement.

seqTC
A **TypeCode** that describes the data type of the elements in a sequence.

arrayTC
A **TypeCode** that describes the data type of the elements of an array.

ptrTC
A **TypeCode** that describes the data type referenced by a pointer.

typename
A string that provides the name of a foreign type.

impCtx
A string that identifies an implementation context where a foreign type is understood.

instSize
A long that holds the size of a foreign type instance. If the size is variable is not known, the value zero should be used.

allOtherTagValues
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.

Return Value

A new **TypeCode** instance, or NULL if the new instance could not be created.

Related Information

TypeCode_alignment Function

TypeCode_copy Function

TypeCode_equal Function

TypeCode_free Function

TypeCode_kind Function

TypeCode_param_count Function

TypeCode_parameter Function

TypeCode_print Function

TypeCode_setAlignment Function

TypeCode_size Function

TypeCode_alignment Function

Supplies the alignment value for a given **TypeCode**.

C Syntax

short TypeCode_alignment ( );

Description

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.

Parameters

tc
The **TypeCode** whose alignment information is desired.

ev
A pointer to an Environment structure.

Return Value

A short integer containing the alignment value.

Related Information

TypeCodeNew Function

TypeCode_equal Function

TypeCode_free Function

TypeCode_kind Function

TypeCode_param_count Function

TypeCode_parameter Function

TypeCode_print Function

TypeCode_setAlignment Function

TypeCode_size Function

TypeCode_copy Function

Creates a new copy of a given **TypeCode**.

C Syntax

TypeCode TypeCode_copy ( );

Description

The **TypeCode_copy** function creates a new copy of a given **TypeCode**. **TypeCodes** are complex data structures whose actual representation is hidden, and may contain internal references to strings and other **TypeCodes**. The copy created by this function is guaranteed not to refer to any previously existing **TypeCodes** or strings, and hence can be used long after the original **TypeCode** is freed or released (**TypeCodes** 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.

Parameters

tc
The **TypeCode** to be copied.

ev
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.

Return Value

A new **TypeCode** with no internal references to any previously existing **TypeCodes** or strings. If a copy cannot be created successfully, the value NULL is returned. No exceptions are raised by this function.

Related Information

TypeCodeNew Function

TypeCode_alignment Function

TypeCode_equal Function

TypeCode_free Function

TypeCode_kind Function

TypeCode_param_count Function

TypeCode_parameter Function

TypeCode_print Function

TypeCode_setAlignment Function

TypeCode_size Function

TypeCode_equal Function

Compares two **TypeCodes** for equality.

C Syntax

```
boolean  TypeCode_equal (
                              TypeCode tc2);
```

Description

The **TypeCode_equal** function can be used to determine if two distinct **TypeCodes** describe the same underlying abstract data type.

Parameters

tc
One of the **TypeCodes** to be compared.

ev
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
The other **TypeCode** to be compared.

Return Value

Returns TRUE (1) if the **TypeCodes** tc and tc2 describe the same data type, with the same alignment. Otherwise, FALSE (0) is returned. No exceptions are raised by this function.

Related Information

TypeCodeNew Function

TypeCode_alignment Function

TypeCode_copy Function

TypeCode_free Function

TypeCode_kind Function

TypeCode_param_count Function

TypeCode_parameter Function

TypeCode_print Function

TypeCode_setAlignment Function

TypeCode_size Function

TypeCode_free Function

Destroys a given **TypeCode** by freeing all of the memory used to represent it.

C Syntax

void TypeCode_free ( );

Description

The **TypeCode_free** function destroys a given **TypeCode** by freeing all of the memory used to represent it. **TypeCodes** obtained from the **TypeCode_copy** or **TypeCodeNew** functions should be freed using **TypeCode_free**. **TypeCodes** 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 **TypeCodes** 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 **TypeCodes**, 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.

Parameters

tc
The **TypeCode** to be freed.

ev
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.

Related Information

TypeCodeNew Function

TypeCode_alignment Function

TypeCode_copy Function

TypeCode_equal Function

TypeCode_kind Function

TypeCode_param_count Function

TypeCode_parameter Function

TypeCode_print Function

TypeCode_setAlignment Function

TypeCode_size Function

TypeCode_kind Function

Categorizes the abstract data type described by a **TypeCode**.

C Syntax

```
TCKind  TypeCode_kind ( );
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};
```

Description

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.

Table 2, **TypeCode information per TCKind category**, 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**.

Parameters

tc
The **TypeCode** whose **TCKind** categorization is requested.

ev
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.

Return Value

Returns one of the enumerators listed in the **TCKind** enumeration shown above. No exceptions are raised by this function.

Related Information

TypeCodeNew Function

TypeCode_alignment Function

TypeCode_copy Function

TypeCode_equal Function

TypeCode_free Function

TypeCode_param_count Function

TypeCode_parameter Function

TypeCode_print Function

TypeCode_setAlignment Function

TypeCode_size Function

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	The name of the **struct** member.
		TypeCode	The type of the **struct** member.
tk_union	3N+2	**string**	The name of the **union**.
		TypeCode	The type of the discriminator.
		----- next 3 repeat for each member -----	
		long	The label value
		string	The name of the member.
		TypeCode	The type of the member.
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**	The type of element in the sequence.
		long	The maximum number of elements or 0.
tk_array	2	**TypeCode**	The type of element in the **array**.
		long	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**	The name of the foreign type.
		string	The implementation context.
		long	The size of an instance.

*Table 2. **TypeCode** information per **TCKind** category*

The **TCKind** values **tk_pointer**, **tk_self** and **tk_foreign** are SOM-unique extensions to the CORBA standard. They are provided to permit **TypeCode**s 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 **TypeCodes** to be created that partially describe non-IDL types. Since these foreign **TypeCodes** 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 "Using tk_foreign TypeCode" on page 352 in *Programmer's Guide for SOM and DSOM* for more information about using foreign **TypeCodes** in SOM IDL files.

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_param_count Function

Obtains the number of parameters available in a given **TypeCode**.

C Syntax

long TypeCode_param_count ( );

Description

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 the table given in the description of the **TypeCode_kind** function.

Parameters

tc
The **TypeCode** whose parameter count is desired.

ev
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.

Return Value

Returns the actual number of parameters associated with the given **TypeCode**, in accordance with the table shown in the **TypeCode_kind** description. No exceptions are raised by this function.

Related Information

TypeCodeNew Function

TypeCode_alignment Function

TypeCode_copy Function

TypeCode_equal Function

TypeCode_free Function

TypeCode_kind Function

TypeCode_parameter Function

TypeCode_print Function

TypeCode_setAlignment Function

TypeCode_size Function

TypeCode_parameter Function

Obtains a specified parameter from a given **TypeCode**.

C Syntax

any TypeCode_parameter (
long *index*);

Description

The **TypeCode_parameter** function can be used to obtain any of the parameters contained in a given **TypeCode**. Refer to the table shown in the description of the **TypeCode_kind** function 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 2, **TypeCode information per TCKind category**, under the **TypeCode_kind** function).

Parameters

tc
The **TypeCode** whose parameter is desired.

ev
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
he 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.

Return Value

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.

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);
}
```

Related Information

TypeCodeNew Function

TypeCode_alignment Function

TypeCode_copy Function

TypeCode_equal Function

TypeCode_free Function

TypeCode_kind Function

TypeCode_param_count Function

TypeCode_print Function

TypeCode_setAlignment Function

TypeCode_size Function

TypeCode_print Function

Writes all of the information contained in a given **TypeCode** to “stdout”.

C Syntax

void TypeCode_print ( );

Description

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.

Parameters

tc
The **TypeCode** to be examined.

ev
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.

Related Information

TypeCodeNew Function

TypeCode_alignment Function

TypeCode_copy Function

TypeCode_equal Function

TypeCode_free Function

TypeCode_kind Function

TypeCode_param_count Function

TypeCode_parameter Function

TypeCode_setAlignment Function

TypeCode_size Function

TypeCode_setAlignment Function

Overrides the default alignment value associated with a given **TypeCode.**

C Syntax

void TypeCode_setAlignment (short *alignment*);

Description

The **TypeCode_setAlignment** function overrides the default alignment value associated with a given **TypeCode**.

Parameters

tc
The **TypeCode** to receive the new alignment value.

ev
A pointer to an Environment structure.

alignment
A short integer that specifies the alignment value.

Related Information

TypeCodeNew Function

TypeCode_alignment Function

TypeCode_equal Function

TypeCode_free Function

TypeCode_kind Function

TypeCode_param_count Function

TypeCode_parameter Function

TypeCode_print Function

TypeCode_size Function

TypeCode_size Function

Provides the minimum size of an instance of the abstract data type described by a given **TypeCode**.

C Syntax

long TypeCode_size ( );

Description

The **TypeCode_size** function is used to obtain the minimum 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.

Parameters

tc
The **TypeCode** whose instance size is desired.

ev
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.

Return Value

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.

Related Information

TypeCodeNew Function

TypeCode_alignment Function

TypeCode_copy Function

TypeCode_equal Function

TypeCode_free Function

TypeCode_kind Function

TypeCode_param_count Function

TypeCode_parameter Function

TypeCode_print Function

TypeCode_setAlignment Function

Chapter 4. Metaclass Framework Classes

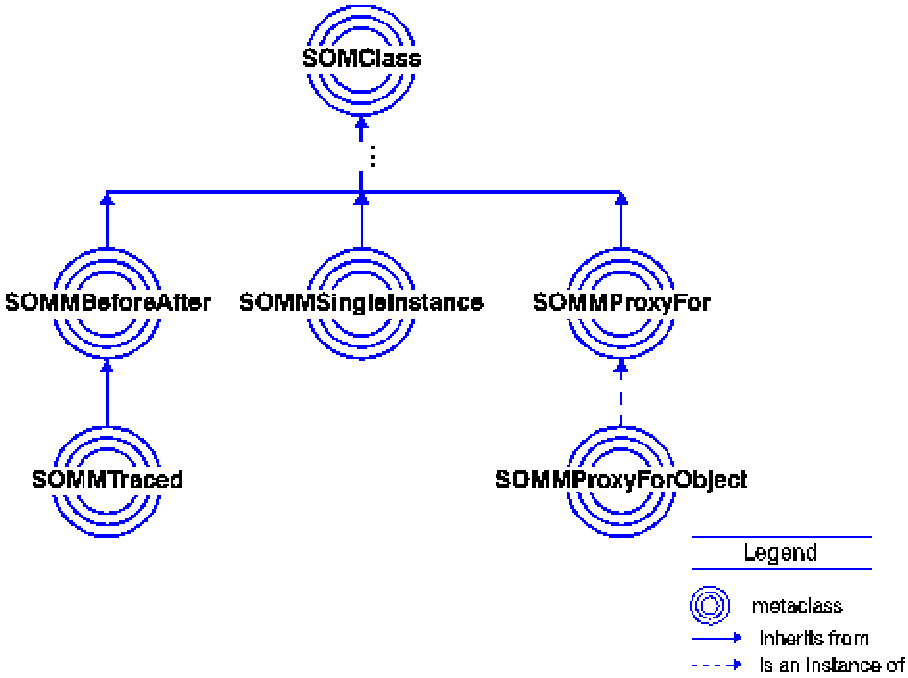

Figure 4. Metaclass Framework Class Organization

SOMMBeforeAfter Metaclass

SOMMBeforeAfter is a metaclass that defines two methods (**sommBeforeMethod** and **sommAfterMethod**) which are invoked 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.

Note: The **somDefaultInit** and **somFree** methods are among the 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 using the object in the **sommBeforeMethod** before **somDefaultInit** has executed, when the object is not yet fully initialized. Second, the implementation must guard against using the object in **sommAfterMethod** after **somFree**, at which time the object no longer exists.

SOMMBeforeAfter is thread-safe.

File Stem

sombacls

New Methods

sommAfterMethod Method

sommBeforeMethod Method

Overriding Methods

somDefaultInit Method

somFree Method

sommAfterMethod Method

Specifies a method that is automatically called after execution of each client method.

IDL Syntax

void sommAfterMethod (
in SOMObject *object,*
in somId *methodId,*
in void ****returnedvalue*,**
in va_list *ap*);

Description

sommAfterMethod specifies a method that is automatically called after execution of each client method and is not directly called by the user. **sommAfterMethod** is introduced in the **SOMMBeforeAfter** metaclass. The default implementation does nothing. To define the desired after method, **sommAfterMethod** must be overridden in a metaclass that is a child of the **SOMMBeforeAfter** metaclass.

The **somFree Method** 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** using the object after **somFree**, at which time the object no longer exists.

Parameters

Refer to the Example's diagram for further clarification of these arguments.

receiver
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.

ev
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
A pointer to the instance of the receiver on which the method is invoked.

methodId
The SOM ID of the method (such as, `myMethod`) that was invoked.

returnedvalue
A pointer to the value returned by invoking the method (`myMethod`) on an object.

ap
The list of input arguments to the method (`myMethod`).

Example

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**.

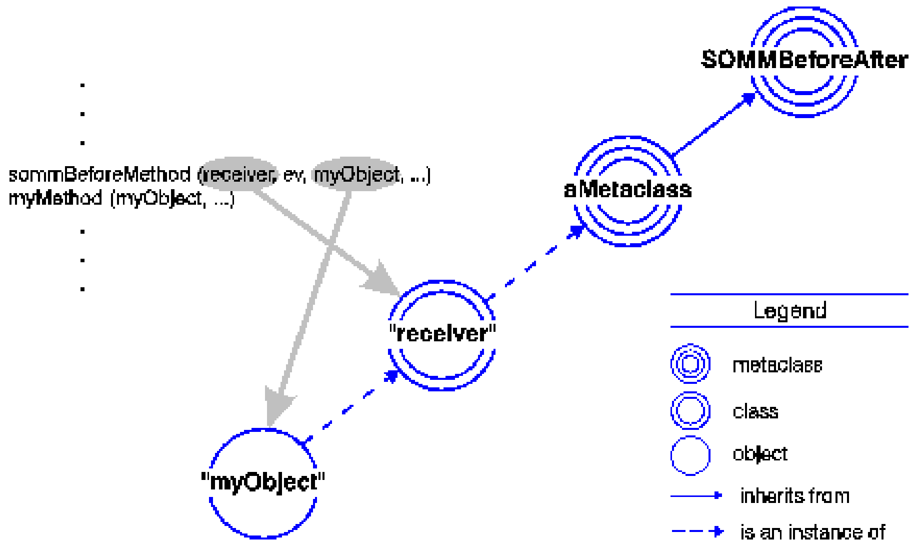

Figure 5. sommAfterMethod Invocation on `myMethod` *on* `MyObject`.

Original Class

SOMMBeforeAfter Metaclass

Related Information

sommBeforeMethod Method

sommBeforeMethod Method

Specifies a method that is automatically called before execution of each client method.

IDL Syntax

boolean sommBeforeMethod (
in SOMObject *object,*
in somId *methodId,*
in va_list *ap*);

Description

sommBeforeMethod a method that is automatically called before execution of each client method and cannot be called directly by the user. To define the desired before method, **sommBeforeMethod** must be overridden in a metaclass that is a subclass of **SOMMBeforeAfter.** The default implementation does nothing until it is overridden.

The **somDefaultInit Method** 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** using the object before the **somDefaultInit** method has executed and the object is not yet fully initialized.

Parameters

Refer to the Example's diagram for further clarification of these arguments.

receiver
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.

ev
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
A pointer to the instance of the receiver on which the method is invoked.

methodId
The SOM ID of the method (`myMethod`) that was invoked.

ap
The list of input arguments to the method (`myMethod`).

Return Value

A boolean that indicates if 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 design eliminates the complexity of communicating to the **sommAfterMethod** that the **sommBeforeMethod** returned FALSE. CORBA specifies that TRUE is 1.

Example

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**

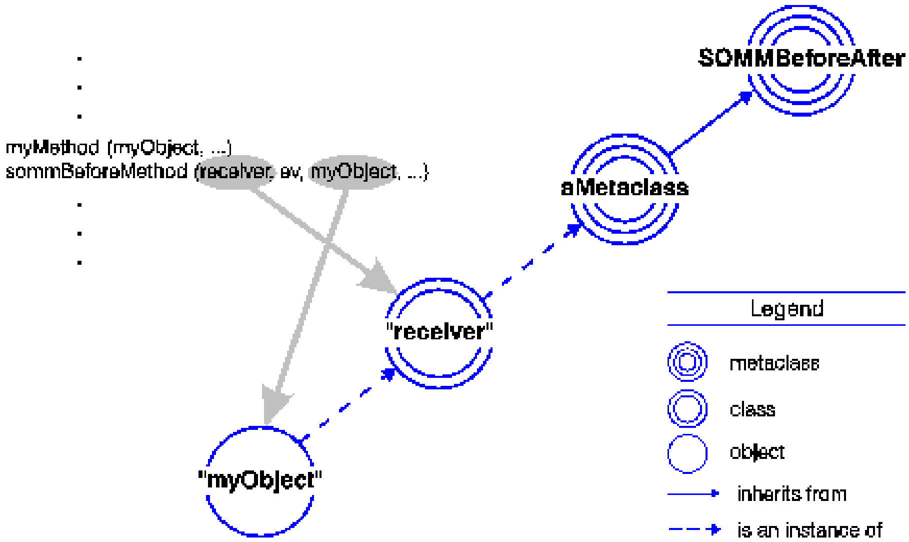

Figure 6. sommBeforeMethod Invocation on `myMethod` on `MyObject`.

Original Class

SOMMBeforeAfter Metaclass

Related Information

sommAfterMethod Method

SOMMProxyFor Metaclass

SOMMProxyFor is the metaclass for creating proxy base classes; it may not be subclassed. Normally, such a metaclass is not made public; however, there is an exception in this case, since the method **sommMakeProxyClass** forces the metaclass to be public. Note that the SOM kernel does enforce the restriction that **SOMMProxyFor** may not be subclassed.

File Stem

somproxy

New Methods

sommMakeProxyClass Method

Overriding Methods

somInitMIClass

somClassReady Method

sommMakeProxyClass Method

Dynamically creates the proxy class corresponding to a target class.

IDL Syntax

SOMClass sommMakeProxyClass (
in SOMClass *targetClass,*
in string *className*);

Description

sommMakeProxyClass invocation creates the proxy class for a specific target class; the resulting class is registered with the class manager.

Parameters

receiver
A pointer to a subclass of **SOMMProxyForObject** that will be the base class for a proxy**.**

targetClass
The class name of the object for which a proxy is to be created.

className
The name of the resulting proxy class. If this parameter is NULL or the empty string, a name is constructed of the form ProxyFor*name-of-targetClass*.

Example

Using object `Lassie` and **SOMMProxyFor**, **SOMMProxyForObject** and `Dog` classes.

```
ProxyForDog  = _sommMakeProxyClass (_SOMMProxyForObject,_Dog,NULL);
proxyForLassie = _somNew(ProxyForDog);
__set_sommTarget( proxyForLassie, Lassie );
```

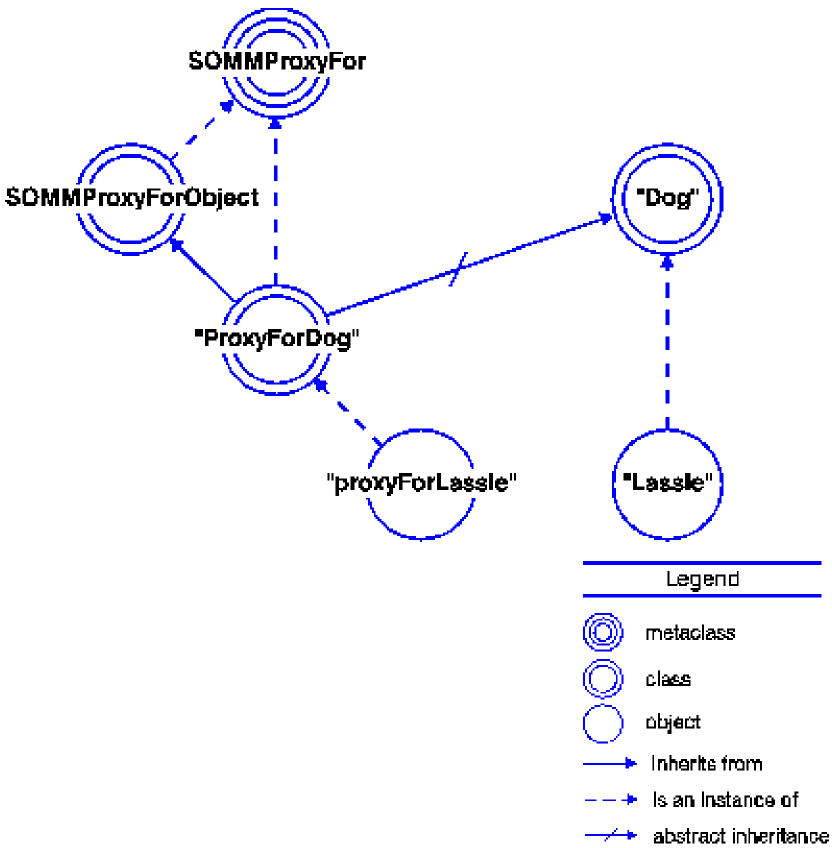

Figure 7. sommMakeProxyClass Method

Original Class

SOMMProxyFor Metaclass

SOMMProxyForObject Class

SOMMProxyForObject is the base class for creating proxy classes. This base class creates transparent proxies. That is, all methods are forwarded from the proxy to the target objects except the **somGetClass, somGetClassName, somGetSize, somIsA, somIsInstanceOf** and **somRespondsTo** methods.

The diagram below illustrates the terminology for discussing proxies. It can be created by the code given in the example of method sommMakeProxyClass. The wavy arrow indicates abstract inheritance.

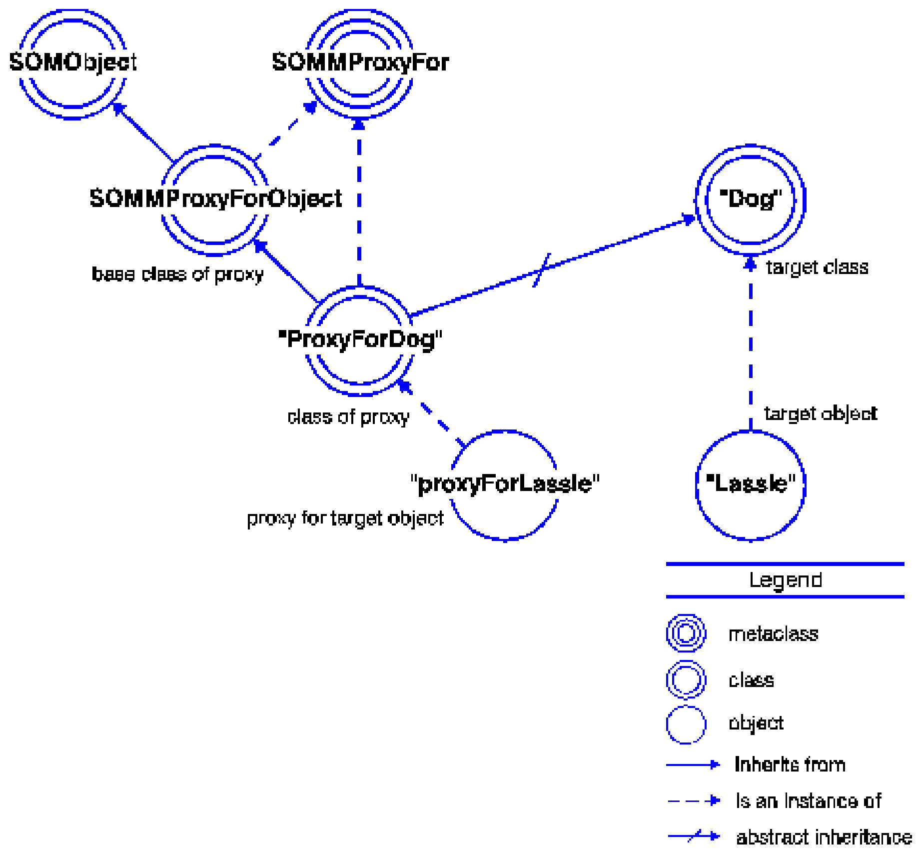

Figure 8. Terminology for discussing Proxies.

The **SOMMProxyForObject** class can be subclassed to create specialized proxies. All methods of a subclass are also forwarded unless they are overridden. Overridden methods can be forwarded with a call to the new method **sommProxyDispatch**. **sommProxyDispatch** can be overridden to enable special ways of dispatching.

When an ordinary object is a **SOMMProxyForObject**, the target object is set with the method **_set_sommProxyTarget**.

Both **somFree** and **somDestruct** are forwarded from the proxy to the target. After each, the proxy object is freed. Because **somDestruct** is forwarded by default proxies, you must be careful if target objects are not allocated from the heap (that is, with **somNew**). There are two techniques that can be used to handle this situation. The first is to ensure that

somDestruct does the appropriate action when invoked at the target object. The second is to create a specialized proxy that does not forward **somDestruct**.

If a base proxy class is statically created, you can compose the base proxy with a Before/After Metaclass. For example, the following IDL creates the figure below in a manner analoguous to the example for **sommMakeProxyClass**.

```
interface TracedProxyForObject : SOMMProxyForObject {
    interface {
        metaclass = SOMMTraced;
    };
};
```

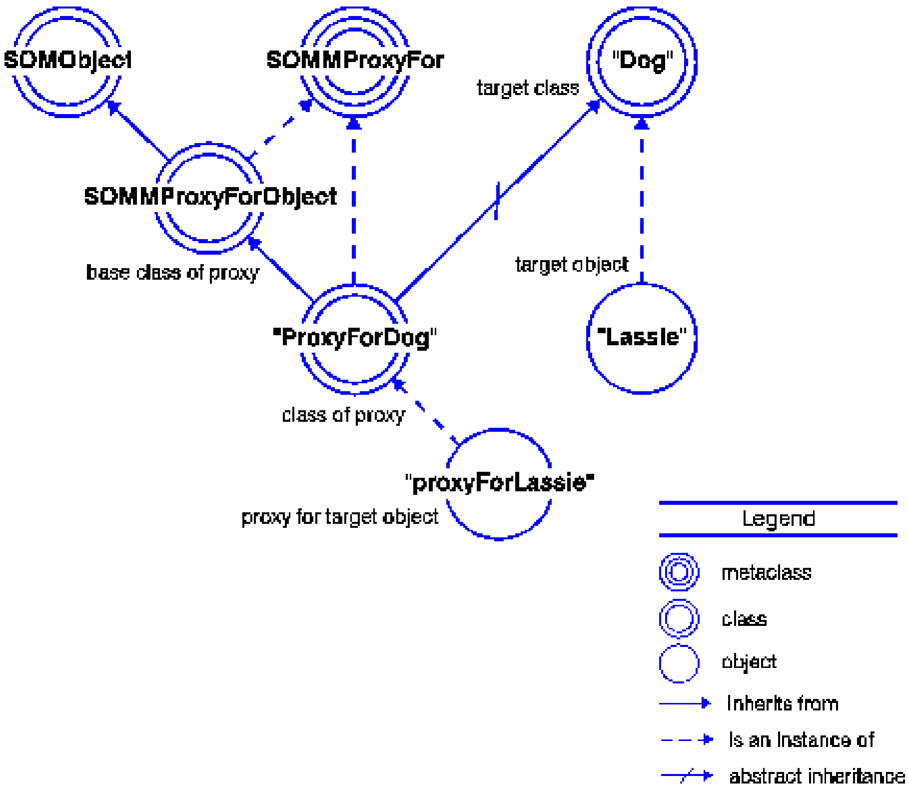

Figure 9. Proxy Tracing for Forwarded Methods.

Note that, in the case of proxy creation, the metaclass of the target class is not used in building the proxy class. For example, consider the figure below. `ProxyForTracedDog` is different from `TracedProxyForDog` because tracing occurs only for methods that are forwarded. This happens because a proxy class only inherits the interface of the target class, not its metaclass constraints.

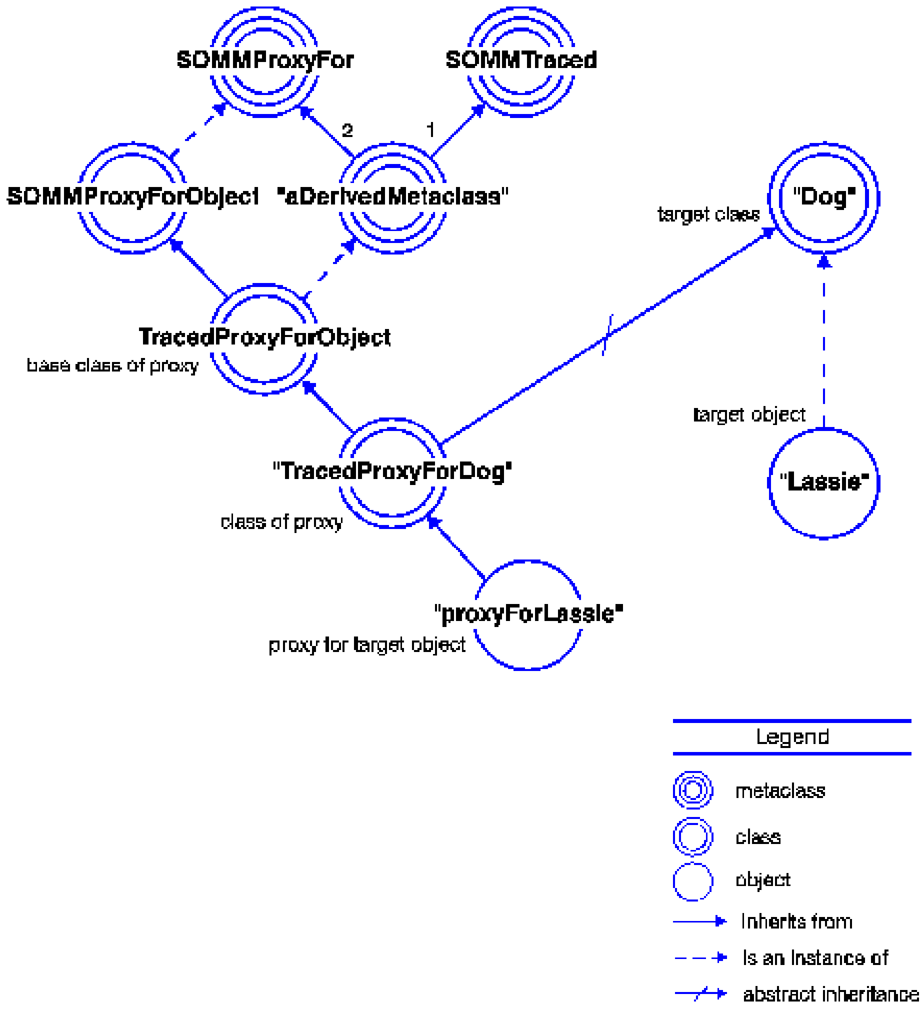

Figure 10. Proxy Tracing for non-Forwarded Methods.

File Stem

somproxy

Attributes

SOMObject sommProxyTarget
This attribute is a pointer to the target object of the proxy

New Methods

sommProxyDispatch Method

_set_sommProxyTarget

_get_sommProxyTarget

Overriding Methods

somDestruct Method

somFree Method

sommProxyDispatch Method

Forwards methods from a proxy to a target object.

IDL Syntax

boolean sommProxyDispatch (
out somToken returnBufferPtr,
in somDispatchInfo dispatchInfo,
in va_list ap);

Description

This method forwards method invocations to the target object. That is, if a method is invoked on a proxy, the invocation is redispatched to **sommProxyDispatch**, which then forwards the invocation to the target, which must be a local object (in the same address space). The **sommProxyDispatch** method can be specialized (overridden) to forward invocations in different ways (for example, to forward invocations to another address space).

This method is not usually invoked by an application, because **sommProxyDispatch** receives the redispatch by default. However, if a method is overridden in a proxy, only the override is invoked. In order to forward a method invocation from an override, **sommProxyDispatch** should be invoked.

Parameters

receiver
A pointer to a **SOMMProxyForObject** object representing an object (called the target object) that responds to the method described in *dispatchInfo*.

returnBufferPtr
A pointer to the value returned when the method described in *dispatchInfo* is invoked on the target object.

dispatchInfo
A pointer to a structure of the following type:

```
typedef struct {
    somMethodData*  md;
    somMethodProc*  dispatchFcn;
    ... arbitrary remaining structure known to dispatchFcn
} somDispatchInfo;
```

ap
The list of input arguments to the method to be forwarded to the target object.

Return Value

A boolean representing whether or not the method was successfully dispatched.

Example

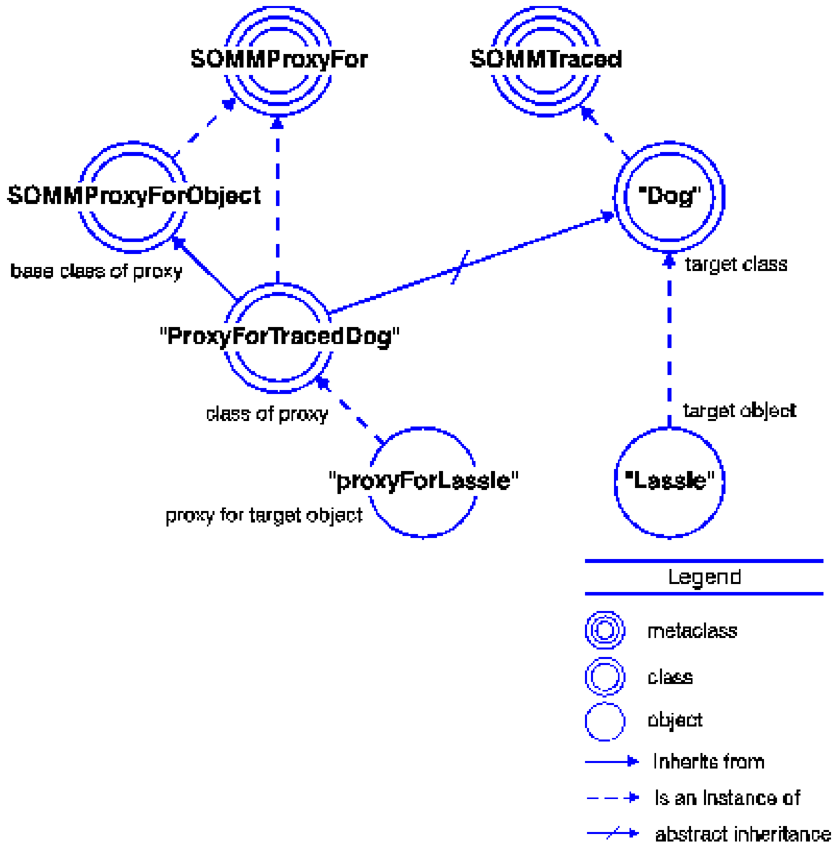

Figure 11. somProxyDispatch Creation Relationship

The relationships shown in the figure above can be created as follows:

```
SOMClass ProxyForDog;
SOMObject proxyForLassie;
Dog Lassie;
Lassie = DogNew();
ProxyForDog = _sommMakeProxyClass
                    (_SOMMProxyForObject, _Dog, NULL );
proxyForLassie = _somNew( ProxyForDog );
__set_sommProxyTarget( proxyForLassie, Lassie );
```

Following this, any method invoked on `proxyForLassie` is forwarded as an invocation on `Lassie`. That is,

```
_aMethod( proxyForLassie, ... );
```

causes the following invocation (which is implemented with **somDispatch** inherited from `Dog`):

```
_aMethod( Lassie, ... );
```

Original Class

SOMMProxyForObject Class

Related Information

sommMakeProxyClass Method

SOMMSingleInstance Metaclass

SOMMSingleInstance can be specified as the metaclass when a class implementor is 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.

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.

SOMMSingleInstance is thread-safe.

File Stem

snglicls

Base Class

SOMClass Class

Metaclass

SOMClass Class

Ancestor Classes

SOMClass Class

SOMObject Class

New Methods

sommGetSingleInstance Method

sommGetSingleInstance Method

Gets the one instance of a specified class for which only a single instance can exist.

IDL Syntax

SOMObject sommGetSingleInstance ( );

Description

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 offers an advantage, however, 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.)

Parameters

receiver
A pointer to a class object whose metaclass is **SOMMSingleInstance**.

ev
A pointer where the method can return exception information if an error is encountered.

Return Value

The **sommGetSingleInstance** method returns a pointer to the single instance of the specified class.

Example

Suppose the class `XXX` is an instance of **SOMMSingleInstance**; then the following C code fragment passes the assertions.

```
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.**

Original Class

SOMMSingleInstance Metaclass

SOMMTraced Metaclass

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 prints 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.

File Stem

somtrcls

Base Class

SOMMBeforeAfter Metaclass

Ancestor Classes

SOMMBeforeAfter Metaclass

SOMClass Class

SOMObject Class

Attributes

boolean sommTraceIsOn
This attribute indicates whether or not tracing is turned on for a class. This gives dynamic control over the trace facility.

Overriding Methods

sommBeforeMethod Method

sommAfterMethod Method

Index

Printed in U.S.A.

.