

C++ API Reference

**Borland
VisiBroker[®] 7.0**

Borland[®]

Borland Software Corporation
20450 Stevens Creek Blvd., Suite 800
Cupertino, CA 95014 USA
www.borland.com

Refer to the file [deploy.html](#) for a complete list of files that you can distribute in accordance with the License Statement and Limited Warranty.

Borland Software Corporation may have patents and/or pending patent applications covering subject matter in this document. Please refer to the product CD or the About dialog box for the list of applicable patents. The furnishing of this document does not give you any license to these patents.

Copyright 1992–2006 Borland Software Corporation. All rights reserved. All Borland brand and product names are trademarks or registered trademarks of Borland Software Corporation in the United States and other countries. All other marks are the property of their respective owners.

Microsoft, the .NET logo, and Visual Studio are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries.

For third-party conditions and disclaimers, see the Release Notes on your product CD.

VB70C++APIRef
March 2006
PDF

Contents

Chapter 1		
Introduction to Borland VisiBroker	1	
VisiBroker Overview	1	
VisiBroker features	2	
VisiBroker Documentation	2	
Accessing VisiBroker online help topics in the standalone Help Viewer	3	
Accessing VisiBroker online help topics from within the VisiBroker Console	3	
Documentation conventions	4	
Platform conventions	4	
Contacting Borland support	4	
Online resources	5	
World Wide Web	5	
Borland newsgroups	5	
Chapter 2		
Generated interfaces and classes	7	
Generated interfaces and classes overview	7	
<Interface_name>	7	
<Interface_name>ObjectWrapper	8	
POA<class_name>	8	
tie<class_name>	8	
<class_name>_var	8	
Chapter 3		
Core interfaces and classes	9	
PortableServer::AdapterActivator	9	
IDL Definition	9	
PortableServer::AdapterActivator methods	10	
BindOptions	10	
Deprecated as of VisiBroker 4.x	10	
Include file	10	
BindOptions members	10	
BOA	11	
Deprecated as of VisiBroker 4.0	11	
Include file	11	
CORBA::BOA methods	12	
VisiBroker extensions to CORBA::BOA	15	
CompletionStatus	16	
IDL Definition	16	
CompletionStatus members	16	
Context	16	
Include file	16	
Context methods	16	
PortableServer::Current	18	
IDL Definition	18	
PortableServer::Current methods	19	
Exception	19	
Include file	19	
Object	19	
Include file	19	
CORBA::Object methods	20	
VisiBroker extensions to CORBA::Object	22	
ORB	24	
Include file	24	
CORBA::ORB methods	24	
VisiBroker extensions to CORBA::ORB	29	
PortableServer::POA	30	
PortableServer::POA methods	30	
PortableServer::POAManager	38	
Include file	39	
PortableServer::POAManager methods	40	
Principal	41	
Include file	41	
Principal methods	41	
PortableServer::RefCountServantBase	41	
Include file	41	
PortableServer::RefCountServantBase methods	42	
PortableServer::ServantActivator	42	
Include file	42	
PortableServer::ServantActivator methods	42	
PortableServer::ServantBase	43	
Include file	43	
PortableServer::ServantBase methods	43	
PortableServer::ServantLocator	44	
Include file	44	
PortableServer::ServantLocator methods	44	
PortableServer::ServantManager	45	
Include file	45	
SystemException	45	
Include file	46	
SystemException methods	46	
Chapter 4		
Dynamic interfaces and classes	49	
Include file	49	
Any methods	49	
Insertion operators	50	
Extraction operators	51	
ContextList	51	
ContextList methods	51	
DynamicImplementation	53	
DynamicImplementation methods	53	
DynAny	53	
Include file	54	
Important usage restrictions	54	
DynAny methods	54	
Extraction methods	55	
Insertion methods	56	
DynAnyFactory	57	
DynAnyFactory methods	57	
DynArray	57	
Important usage restrictions	57	
DynArray methods	57	
DynEnum	58	
Important usage restrictions	58	
DynEnum methods	58	

DynSequence	59	EnumDef	91
Important usage restrictions	59	EnumDef methods	91
DynSequence methods	59	ExceptionDef	92
DynStruct	60	ExceptionDef methods	92
Important usage restrictions	60	ExceptionDescription	92
DynStruct methods	60	ExceptionDescription members	92
DynUnion	61	FixedDef	93
Important usage restrictions	61	Methods	93
DynUnion methods	61	FullInterfaceDescription	93
Environment	62	FullInterfaceDescription members	93
Include file	62	FullValueDescription	94
Environment methods	62	Variables	94
ExceptionList	63	IDLType	95
ExceptionList methods	63	Include file	95
NamedValue	65	IDLType methods	95
Include file	65	InterfaceDef	96
NamedValue methods	65	Include file	96
NVList	66	InterfaceDef methods	97
Include file	66	InterfaceDescription	98
NVList methods	66	InterfaceDescription members	98
Request	69	IObject	99
Include file	69	Include file	99
Request methods	69	IObject methods	99
ServerRequest	72	ModuleDef	99
Include file	72	ModuleDescription	99
ServerRequest methods	72	ModuleDescription members	99
TCKind	74	NativeDef	100
TypeCode	75	OperationDef	100
Include file	75	Include file	100
TypeCode constructors	75	OperationDef methods	101
TypeCode methods	75	OperationDescription	102
		OperationDescription members	102
Chapter 5		OperationMode	103
Interface repository interfaces		OperationMode values	103
and classes	79	ParameterDescription	103
AliasDef	79	ParameterDescription members	103
AliasDef methods	79	ParameterMode	103
ArrayDef	80	ParameterMode values	104
ArrayDef methods	80	PrimitiveDef	104
AttributeDef	80	PrimitiveDef methods	104
AttributeDef methods	80	PrimitiveKind	104
AttributeDescription	81	PrimitiveKind values	104
AttributeDescription members	81	Repository	105
AttributeMode	82	Include file	105
AttributeMode values	82	Repository methods	105
ConstantDef	82	SequenceDef	106
ConstantDef methods	82	SequenceDef methods	107
ConstantDescription	83	StringDef	107
ConstantDescription members	83	StringDef methods	107
Contained	83	StructDef	107
Include file	84	StructDef methods	108
Contained methods	84	StructMember	108
Container	85	StructMember methods	108
Include file	86	TypedefDef	108
Container methods	86	TypeDescription	108
DefinitionKind	90	TypeDescription members	109
DefinitionKind values	90	UnionDef	109
Description	91	UnionDef methods	109
Description members	91	UnionMember	110
		UnionMember members	110

ValueBoxDef	110
Methods	110
ValueDef	111
Methods	111
ValueDescription	113
Values	113
WstringDef	114
WStringDef methods.	114

Chapter 6

Activation interfaces and classes 115

ImplementationStatus	115
Include file	115
ImplementationStatus members.	115
OAD.	116
Include file	117
OAD methods	117
ObjectStatus	120
Include file	120
ObjectStatus members	120
ObjectStatusList	121
Include file	121
ObjectStatusList methods	121

Chapter 7

Naming Service (VisiNaming) interfaces and classes 123

NamingContext	123
NamingContext methods	124
NamingContextExt	128
NamingContextExt methods	128
Binding and BindingList.	129
BindingIterator	130
BindingIterator methods	130
NamingContextFactory	131
Methods	131
ExtendedNamingContextFactory	132
Methods	132

Chapter 8

Event service interfaces and classes 133

ConsumerAdmin	133
IDL definition.	133
ConsumerAdmin methods.	133
EventChannel	134
Methods	134
EventChannelFactory	135
IDL definition.	135
EventChannelFactory methods	135
ProxyPullConsumer.	136
IDL definition.	136
ProxyPushConsumer	136
IDL definition.	136
ProxyPullSupplier	136
IDL definition.	137
ProxyPushSupplier	137
IDL definition.	137
PullConsumer	137
IDL definition.	137

PushConsumer	138
IDL definition	138
PullSupplier	138
IDL definition	138
PullSupplier methods.	138
PushSupplier	139
IDL definition	139
SupplierAdmin	139
IDL definition	139

Chapter 9

Server Manager Interfaces and Classes 141

The Container Interface	141
The Container Interface	141
Methods related to property manipulation and queries	141
Methods related to operations.	142
Methods related to children containers	143
Methods related to storage	144
The Storage Interface	144
Storage Interface Methods for C++.	144

Chapter 10

Transaction Service interfaces and classes 147

CosTransactions and VISTransactions modules	148
Looking at the CosTransactions module	148
Data types.	148
Structures	149
Exceptions.	150
Looking at the VISTransactions module	151
Current interface	151
Choosing a Current interface	151
Obtaining a Current object reference.	152
Using the Current object reference.	153
Is your VisiTransact Transaction Service instance available?	153
Checked behavior	153
Current methods	153
TransactionalObject interface	165
TransactionFactory interface.	165
TransactionFactory methods	166
Control interface	168
Control methods	169
Terminator interface	170
Terminator methods	170
Coordinator interface	172
Coordinator methods	173
RecoveryCoordinator interface	180
RecoveryCoordinator methods.	180
Resource interface.	181
Resource methods	182
Synchronization interface	185
Synchronization methods.	186
VISTransactionService class	188
VISTransactionService methods	188

Parameter	231
Include file	231
Members	231
ParameterList	231
Include file	231
PolicyFactory	232
Include file	232
PolicyFactory Method	232
RequestInfo	232
Include file	232
RequestInfo methods	233
ServerRequestInfo	235
Include file	236
ServerRequestInfo methods	236
ServerRequestInterceptor	238
Include file	238
ServerRequestInterceptor methods	238

Chapter 13

5.x Interceptor and object wrapper interfaces and classes

241

Introduction	241
InterceptorManagers	242
IOR templates	242
InterceptorManager	242
InterceptorManagerControl	242
Include file	243
InterceptorManagerInterceptor method	243
BindInterceptor	243
Include file	243
BindInterceptor methods	243
BindInterceptorManager	244
Include file	244
BindInterceptorManager method	244
ClientRequestInterceptor	245
Include file	245
ClientRequestInterceptor methods	245
ClientRequestInterceptorManager	246
Include file	246
ClientRequestInterceptorManager methods	247
POALifeCycle Interceptor	247
Include file	247
POALifeCycleInterceptor methods	247
POALifeCycleInterceptorManager	248
Include file	248
POALifeCycleInterceptorManager method	248
ActiveObjectLifeCycleInterceptor	248
Include file	248
ActiveObjectLifeCycleInterceptor methods	248
ActiveObjectLifeCycleInterceptorManager	249
Include file	249
ActiveObjectLifeCycleInterceptorManager method	249
ServerRequestInterceptor	249
Include file	249
ServerRequestInterceptor methods	250
ServerRequestInterceptorManager	251
Include file	251
ServerRequestInterceptorManager method	251

IORCreationInterceptor	251
Include file	251
IORInterceptor method	252
IORCreationInterceptorManager	252
Include file	252
IORCreationInterceptorManager method	252
Closure	252
ExtendedClosure	253
VISClosure	253
Include file	253
VISClosure members	253
VISClosureData	254
VISClosureData methods	254
ChainUntypedObjectWrapperFactory	254
Include file	254
ChainUntypedObjectWrapperFactory methods	254
UntypedObjectWrapper	256
Include file	256
UntypedObjectWrapper methods	256
UntypedObjectWrapperFactory	257
Include file	257
UntypedObjectWrapperFactory constructor	257
UntypedObjectWrapperFactory methods	257

Chapter 14

Quality of Service interfaces and classes

259

CORBA::PolicyManager	259
IDL definition	259
Methods	260
CORBA::Object	260
IDL definition	260
Methods	261
Messaging::RebindPolicy	262
IDL definition	262
Policy values	262
QoSExt::DeferBindPolicy	263
IDL definition	263
QoSExt::RelativeConnectionTimeoutPolicy	263
IDL definition	264
Messaging::RelativeRequestTimeoutPolicy	264

Chapter 15

IOP and IIOF interfaces and classes

265

GIOP::MessageHeader	265
MessageHeader members	265
GIOP::CancelRequestHeader	266
CancelRequestHeader members	266
GIOP::LocateReplyHeader	266
LocateReplyHeader members	266
GIOP::LocateRequestHeader	267
LocateRequestHeader members	267
GIOP::ReplyHeader	267
Include file	267
ReplyHeader members	267
GIOP::RequestHeader	268
Include file	268
RequestHeader members	268

IOP::ProfileBody	269
ProfileBody members	269
IOP::IOR	269
Include file	269
IOR members	270
IOP::TaggedProfile	270
TaggedProfile members	270

Chapter 16
Marshal buffer interfaces
and classes **271**

CORBA::MarshalInBuffer	271
Include file	271
CORBA::MarshalInBuffer constructors/ destructors	272
CORBA::MarshalInBuffer methods	272
CORBA::MarshalInBuffer operators	275
CORBA::MarshalOutBuffer	275
Include file	275
CORBA::MarshalOutBuffer constructors/ destructors	276
CORBA::MarshalOutBuffer methods	276
CORBA::MarshalOutBuffer operators	278

Chapter 17
Location service interfaces
and classes **279**

Agent	279
IDL definition	280
Include file	280
Agent methods	280
Desc	284
IDL definition	284
Desc members	284
Fail	285
Fail members	285
TriggerDesc	285
IDL definition	285
TriggerDesc members	286
TriggerHandler	286
IDL definition	286
Include file	286
TriggerHandler methods	286
<type>Seq	287
<type>Seq methods	287
<type>SeqSeq	288
<type>SeqSeq methods	288

Chapter 18
Initialization interfaces
and classes **289**

VISInit	289
Include file	289
VISInit constructors/destructors	289
VISInit methods	290

Chapter 19
Real-Time CORBA interfaces
and classes **291**

Introduction	291
Include file	291
RTCORBA::Current	292
RTCORBA::Current Creation and Destruction	292
IDL definition	292
RTCORBA::Current methods	292
RTCORBA::Mutex	293
Mutex Creation and Destruction	293
IDL definition	293
RTCORBA::Mutex Methods	294
RTCORBA::NativePriority	294
IDL definition	294
RTCORBA::Priority	295
IDL definition	295
RTCORBA::PriorityMapping	295
PriorityMapping Creation and Destruction	296
IDL definition	296
PriorityMapping Methods	296
RTCORBA::PriorityModel	297
RTCORBA::PriorityModelPolicy	298
IDL definition	298
RTCORBA::RTORB	298
RTORB Creation and Destruction	298
IDL definition	299
RTORB Methods	299
RTCORBA::ThreadpoolId	301
IDL definition	301
RTCORBA::ThreadpoolPolicy	302
IDL definition	302

Chapter 20
Pluggable Transport Interface
Classes **303**

VISPTTransConnection	303
Include file	303
VISPTTransConnection methods	303
VISPTTransConnectionFactory	306
Include file	306
VISPTTransConnectionFactory methods	307
VISPTTransListener	307
Include file	307
VISPTTransListener methods	307
VISPTTransListenerFactory	308
Include file	308
VISPTTransListenerFactory methods	309
VISPTTransProfileBase	309
Include file	309
VISPTTransProfileBase methods	309
VISPTTransProfileBase members	310
VISPTTransProfileBase base class methods	310
VISPTTransProfileFactory	311
Include file	311
VISPTTransProfileFactory methods	311

VISPTransBridge	312	VISDAppender	318
Include file	312	Include file	318
VISPTransBridge methods	312	VISDAppender methods	319
VISPTransRegistrar	313	VISDLayoutFactory	319
Include file	313	Include file	319
VISPTransRegistrar methods	313	VISDLayoutFactory methods	319
Chapter 21		VISDLayout	320
VisiBroker for C++ Logging	315	Include file	320
VISDLoggerMgr	315	VISDLayout methods	320
Include file	315	VISDConfig	320
VISDLoggerMgr methods	315	Include file	320
VISDLogger	317	LogAppenderConfig structure	321
Include file	317	VISDLogRecord	321
VISDLogger methods	317	Include file	321
VISDAppenderFactory	318	VISDLogRecord methods	321
Include file	318	VISDLogLevel	322
VISDAppenderFactory methods	318	Include file	322
		Level enumeration	322

Index **323**

Introduction to Borland VisiBroker

For the CORBA developer, Borland provides *VisiBroker for Java*, *VisiBroker for C++*, and *VisiBroker for .NET* to leverage the industry-leading VisiBroker Object Request Broker (ORB). These three facets of VisiBroker are implementations of the CORBA 2.6 specification.

VisiBroker Overview

VisiBroker is for distributed deployments that require CORBA to communicate between both Java and non-Java objects. It is available on a wide range of platforms (hardware, operating systems, compilers and JDKs). VisiBroker solves all the problems normally associated with distributed systems in a heterogeneous environment.

VisiBroker includes:

- VisiBroker for Java, VisiBroker for C++, and VisiBroker for .NET, three implementations of the industry-leading Object Request Broker.
- VisiNaming Service, a complete implementation of the Interoperable Naming Specification in version 1.3.
- GateKeeper, a proxy server for managing connections to CORBA Servers behind firewalls.
- VisiBroker Console, a GUI tool for easily managing a CORBA environment.
- Common Object Services such as VisiNotify (implementation of Notification Service Specification), VisiTransact (implementation of Transaction Service Specification), VisiTelcoLog (implementation of Telecom Logging Service Specification), VisiTime (implementation of Time Service Specification), and VisiSecure.

VisiBroker features

VisiBroker offers the following features:

- “Out-of-the-box” security and web connectivity.
- Seamless integration to the J2EE Platform, allowing CORBA clients direct access to EJBs.
- A robust Naming Service (VisiNaming), with caching, persistent storage, and replication for high availability.
- Automatic client failover to backup servers if primary server is unreachable.
- Load distribution across a cluster of CORBA servers.
- Full compliance with the OMG's CORBA 2.6 Specification.
- Integration with the Borland JBuilder integrated development environment.
- Enhanced integration with other Borland products including Borland AppServer.

VisiBroker Documentation

The VisiBroker documentation set includes the following:

- Borland *VisiBroker Installation Guide*—describes how to install VisiBroker on your network. It is written for system administrators who are familiar with Windows or UNIX operating systems.
- Borland *Security Guide*—describes Borland's framework for securing VisiBroker, including VisiSecure for VisiBroker for Java and VisiBroker for C++.
- Borland *VisiBroker for Java Developer's Guide*—describes how to develop VisiBroker applications in Java. It familiarizes you with configuration and management of the Visibroker ORB and how to use the programming tools. Also described is the IDL compiler, the Smart Agent, the Location, Naming and Event Services, the Object Activation Daemon (OAD), the Quality of Service (QoS), the Interface Repository, and the Interface Repository, and Web Service Support.
- Borland *VisiBroker for C++ Developer's Guide*—describes how to develop VisiBroker applications in C++. It familiarizes you with configuration and management of the Visibroker ORB and how to use the programming tools. Also described is the IDL compiler, the Smart Agent, the Location, Naming and Event Services, the OAD, the QoS, Pluggable Transport Interface, RT CORBA Extensions, and Web Service Support.
- Borland *VisiBroker for .NET Developer's Guide*—describes how to develop VisiBroker applications in a .NET environment.
- Borland *VisiBroker for C++ API Reference*—provides a description of the classes and interfaces supplied with VisiBroker for C++.
- Borland *VisiBroker VisiTime Guide*—describes Borland's implementation of the OMG Time Service specification.
- Borland *VisiBroker VisiNotify Guide*—describes Borland's implementation of the OMG Notification Service specification and how to use the major features of the notification messaging framework, in particular, the Quality of Service (QoS) properties, Filtering, and Publish/Subscribe Adapter (PSA).
- Borland *VisiBroker VisiTransact Guide*—describes Borland's implementation of the OMG Object Transaction Service specification and the Borland Integrated Transaction Service components.

- Borland *VisiBroker VisiTelcoLog Guide*—describes Borland's implementation of the OMG Telecom Log Service specification.
- Borland *VisiBroker GateKeeper Guide*—describes how to use the VisiBroker GateKeeper to enable VisiBroker clients to communicate with servers across networks, while still conforming to the security restrictions imposed by web browsers and firewalls.

The documentation is typically accessed through the Help Viewer installed with VisiBroker. You can choose to view help from the standalone Help Viewer or from within a VisiBroker Console. Both methods launch the Help Viewer in a separate window and give you access to the main Help Viewer toolbar for navigation and printing, as well as access to a navigation pane. The Help Viewer navigation pane includes a table of contents for all VisiBroker books and reference documentation, a thorough index, and a comprehensive search page.

Important Updates to the product documentation, as well as PDF versions, are available on the web at <http://www.borland.com/techpubs>.

Accessing VisiBroker online help topics in the standalone Help Viewer

To access the online help through the standalone Help Viewer on a machine where the product is installed, use one of the following methods:

- Windows**
- Choose Start|Programs|Borland Deployment Platform|Help Topics
 - or, open the Command Prompt and go to the product installation `\bin` directory, then type the following command:
- ```
help
```
- UNIX**
- Open a command shell and go to the product installation `/bin` directory, then enter the command:
- ```
help
```
- Tip** During installation on UNIX systems, the default is to not include an entry for `bin` in your `PATH`. If you did not choose the custom install option and modify the default for `PATH` entry, and you do not have an entry for current directory in your `PATH`, use `./help` to start the help viewer.

Accessing VisiBroker online help topics from within the VisiBroker Console

To access the online help from within the VisiBroker Console, choose Help|Help Topics.

The Help menu also contains shortcuts to specific documents within the online help. When you select one of these shortcuts, the Help Topics viewer is launched and the item selected from the Help menu is displayed.

Documentation conventions

The documentation for VisiBroker uses the typefaces and symbols described below to indicate special text:

Table 1.1 Documentation conventions

Convention	Used for
<i>italics</i>	Used for new terms and book titles.
<code>computer</code>	Information that the user or application provides, sample command lines and code.
bold computer	In text, bold indicates information the user types in. In code samples, bold highlights important statements.
[]	Optional items.
...	Previous argument that can be repeated.
	Two mutually exclusive choices.

Platform conventions

The VisiBroker documentation uses the following symbols to indicate platform-specific information:

Table 1.2 Platform conventions

Symbol	Indicates
Windows	All supported Windows platforms.
Win2003	Windows 2003 only
WinXP	Windows XP only
Win2000	Windows 2000 only
UNIX	UNIX platforms
Solaris	Solaris only
Linux	Linux only

Contacting Borland support

Borland offers a variety of support options. These include free services on the Internet where you can search our extensive information base and connect with other users of Borland products. In addition, you can choose from several categories of telephone support, ranging from support on installation of Borland products to fee-based, consultant-level support and detailed assistance.

For more information about Borland's support services or contacting Borland Technical Support, please see our web site at: <http://support.borland.com> and select your geographic region.

When contacting Borland's support, be prepared to provide the following information:

- Name
- Company and site ID
- Telephone number
- Your Access ID number (U.S.A. only)
- Operating system and version
- Borland product name and version
- Any patches or service packs applied

- Client language and version (if applicable)
- Database and version (if applicable)
- Detailed description and history of the problem
- Any log files which indicate the problem
- Details of any error messages or exceptions raised

Online resources

You can get information from any of these online sources:

World Wide Web	http://www.borland.com
Online Support	http://support.borland.com (access ID required)
Listserv	To subscribe to electronic newsletters, use the online form at: http://www.borland.com/products/newsletters

World Wide Web

Check <http://www.borland.com/bes> regularly. The VisiBroker Product Team posts white papers, competitive analyses, answers to FAQs, sample applications, updated software, updated documentation, and information about new and existing products.

You may want to check these URLs in particular:

- http://www.borland.com/products/downloads/download_visibroker.html (updated VisiBroker software and other files)
- <http://www.borland.com/techpubs> (documentation updates and PDFs)
- <http://support.borland.com/entry.jspa?externalID=4273&categoryID=112> (VisiBroker FAQs)
- <http://community.borland.com> (contains our web-based news magazine for developers)

Borland newsgroups

You can participate in many threaded discussion groups devoted to the Borland VisiBroker. Visit <http://www.borland.com/newsgroups> for information about joining user-supported newsgroups for VisiBroker and other Borland products.

Note These newsgroups are maintained by users and are not official Borland sites.

2

Generated interfaces and classes

This section describes classes generated by the VisiBroker for C++ IDL compiler, their uses, and their features.

Generated interfaces and classes overview

The VisiBroker IDL compiler generates classes that make it easier for you to develop client applications and object servers. Many of these generated classes are available for CORBA classes.

- stub classes
- servant classes
- tie classes
- var classes

<Interface_name>

The <interface_name> class is generated for a particular IDL interface and is intended for use by client applications. This class provides all of the methods defined for a particular IDL interface. When a client uses an object reference to invoke methods on the object, the stub methods are actually invoked. The stub methods allow a client operation request to be packaged, sent to the object implementation, and the results to be reflected. This entire process is transparent to the client application.

When a client uses a local object reference to invoke methods on the local object, there is no stub method involved.

Note You should never modify the contents of a stub class generated by the IDL compiler.

<Interface_name>ObjectWrapper

This class does not apply to local interfaces. For non-local interface, this class is used to derive typed object wrappers and is generated for all your non-local interfaces when you invoke the `idl2cpp` command with the `-obj_wrapper` option. For more information about the `-obj_wrapper` option, see “Programmer tools for C++” in the *VisiBroker for C++ Developer's Guide*.

```
static void add(CORBA::ORB_ptr orb, CORBA::ObjectFactory factory,
VISObjectWrapper::Location loc); static void remove(CORBA::ORB_ptr orb,
CORBA::ObjectFactory factory, VISObjectWrapper::Location loc);
```

Removes an un-typed object wrapper from a server application.

Parameter	Description
<code>orb</code>	The ORB the client wishes to use, returned by the <code>ORB_init</code> method.
<code>factory</code>	The factory method for the object wrapper class that you want to remove.
<code>loc</code>	The location of the object wrapper being removed, which should be one of the following values: <code>VISObjectWrapper::Client</code> , <code>VISObjectWrapper::Server</code> , <code>VISObjectWrapper::Both</code>

POA<class_name>

The `_POA_<class_name>` class is an abstract base class generated by the IDL compiler, which is used to derive an object implementation class. Object implementations are usually derived from a servant class, which provides the necessary methods for receiving and interpreting client operation requests.

tie<class_name>

The `_tie_<class_name>` class is generated by the IDL compiler to aid in the creation of delegation implementations. The tie class allows you to create an object implementation that delegates all operation requests to another object. This allows you to use existing objects that you do not wish to inherit from the `CORBA::Object` class.

<class_name>_var

The `<class_name>_var` class is generated for an IDL interface and provides simplified memory management semantics.

Core interfaces and classes

This section describes the VisiBroker for C++ core interfaces and classes.

PortableServer::AdapterActivator

Adapter activators are associated with Portable Object Adapters (POAs). They make it possible for POAs to create child POAs under the following circumstances:

- On demand,
- As a side-effect of receiving a request which names the child POA (or one of its children), or
- When the `find_POA` method is called with an `activate` parameter set to `TRUE`.

For more information about POAs, see “Using POAs” in the *VisiBroker for C++ Developer's*.

IDL Defintion

```
interface AdapterActivator {  
  
    boolean unknown_adapter(in POA parent, in string name);  
  
};
```

PortableServer::AdapterActivator methods

```
CORBA::Boolean unknown_adapter(PortableServer::POA_ptr parent, const char* name);
```

This method is called when the VisiBroker ORB receives a request for an object reference which identifies a target POA that does not exist. The VisiBroker ORB invokes this method once for each POA that must be created in order for the POA to exist (starting with the ancestor POA closest to the root POA).

Parameter	Description
parent	The parent POA associated with the adapter activator on which the method is to be invoked.
name	The name of the POA to be created (relative to the parent).

BindOptions

Deprecated as of VisiBroker 4.x

```
struct BindOptions
```

This structure is used to specify options to the `_bind` method, described in [“Object” on page 19](#). Each process has a global `BindOptions` structure that is used for all `_bind` invocations that do not specify bind options. You can modify the default bind options using the `Object::_default_bind_options` method.

Bind options may also be set for a particular object and will remain in effect for the lifetime of the connection to that object.

Include file

The `corba.h` file should be included when you use this structure.

BindOptions members

```
CORBA::Boolean defer_bind;
```

If set to `TRUE`, the establishment of the connection between client and the object implementation is delayed until the first client operation is issued.

If set to `FALSE`, the `_bind` method should establish the connection immediately.

```
CORBA::Boolean enable_rebind;
```

If set to `TRUE` and the connection is lost, due to a network failure or some other error, the VisiBroker ORB attempts to re-establish a connection to a suitable object implementation.

If set to `FALSE`, no attempt is made to reconnect the client with the object implementation.

`CORBA::Long` `max_bind_tries`;

This member specifies the number of times to retry a bind request when the OAD is busy.

`CORBA::ULong` `send_timeout`;

This member specifies the maximum time in seconds that a client is to block waiting to send an operation request. If the request times out, `CORBA::NO_RESPONSE` exception is raised and the connection to the server is destroyed.

The default value of 0 (zero) indicates that the client is to block indefinitely.

`CORBA::ULong` `receive_timeout`;

This member specifies the maximum time in seconds that a client is to block waiting for a response to an operation request. If the request times out, a `CORBA::NO_RESPONSE` exception is raised and the connection to the server is destroyed.

The default value of 0 (zero) indicates that the client is to block indefinitely.

`CORBA::ULong` `connection_timeout`;

This member specifies the maximum time in seconds that a client is to wait for a connection. If the time specified is exceeded, a `CORBA::NO_IMPLEMENT` exception is raised.

The default value of 0 indicates that the default system time-out for connections is to be used.

BOA

Deprecated as of VisiBroker 4.0

`class` `BOA`

The `BOA` class represents the Basic Object Adaptor and provides methods for creating and manipulating objects and object references. Object servers use the `BOA` to activate and deactivate object implementations and to specify the thread policy they wish to use.

You do not instantiate a `BOA` object. Instead, you obtain a reference to a `BOA` object by invoking the `ORB::BOA_init` method.

Borland VisiBroker provides extensions to the CORBA `BOA` specification which are covered in [“VisiBroker extensions to CORBA::BOA” on page 15](#). These methods provide for the management of connections, threads, and the activation of services.

Include file

Include the file `corba.h` when you use this class.

CORBA::BOA methods

CORBA::Object_ptr **create**(const CORBA::ReferenceData& refData,
extension::CreationImplDef& creationImplDef)

This method registers the specified implementation with the OAD.

Parameter	Description
refData	This parameter is not used, but is provided for compliance with the CORBA specification.
creationImplDef	This pointer's true type is <code>CreationImplDef</code> . It provides the interface name, object name, path name of the executable and the activation policy and other parameters. See "Activation interfaces and classes" on page 115 for a complete discussion of the <code>CreationImplDef</code> class.

void **deactivate_impl**(extension::ImplementationDef_ptr implDefPtr)

This method causes requests to the implementation to be discarded.

The method deactivates the implementation specified by `implDefPtr`. Once this method is called, no further client requests are delivered to the object within this implementation until the objects and implementation are re-activated. To cause the implementation to again accept requests, call `impl_is_ready` or `obj_is_ready`.

Parameter	Description
implDefPtr	This pointer's true type is <code>CreationImplDef</code> . It provides the interface name, object name, path name of the executable and activation policy, along with other parameters.

void **deactivate_obj**(CORBA::Object_ptr objPtr)

This method requests that the BOA deactivate the specified object. Once this method is invoked, the BOA does not deliver any requests to the object until `obj_is_ready` or `impl_is_ready` is invoked.

Parameter	Description
objPtr	A pointer to the object to be deactivated.

void **dispose**(CORBA::Object_ptr objPtr)

This method unregisters the implementation of the specified object from the Object Activation Daemon. Once this method is invoked, all references to the specified object are invalid and any connections to this object implementation are broken. If the object has been allocated, it is the application's responsibility to delete the object.

Note This method is deprecated as of VisiBroker 4.x. You are urged to use the OAD's interface instead.

Parameter	Description
objPtr	A pointer to the object to be unregistered.

```
static CORBA::BOA_ptr _duplicate(CORBA::BOA_ptr ptr)
```

This static method duplicates the BOA pointer that is passed in as a parameter.

Parameter	Description
<code>ptr</code>	A BOA pointer.

```
void exit_impl_ready()
```

This method provides backward compatibility with earlier releases of VisiBroker for C++. It invokes `BOA::shutdown`, described in “[void shutdown\(\)](#)” on page 14, which causes a previous invocation of the `impl_is_ready` method to return. This method cannot be invoked in the context of an active request.

```
CORBA::ReferenceData_ptr get_id(CORBA::Object_ptr objPtr)
```

This method returns the reference data for the specified object. The reference data is set by the object implementation at activation time and is guaranteed to remain constant throughout the life of the object.

Parameter	Description
<code>objPtr</code>	A pointer to the object whose reference data is to be returned.

```
CORBA::Principal_ptr get_principal(CORBA::Object_ptr objPtr,
CORBA::Environment_ptr env=NULL)
```

This method returns the `Principal` object associated with the specified object. This method may only be called by an object implementation during the processing of a client operation request.

Parameter	Description
<code>objPtr</code>	A pointer to the object whose implementation is to be changed.
<code>env</code>	A pointer to the <code>Environment</code> object associated with this <code>Principal</code> .

```
void impl_is_ready(const char *service_name, extension::Activator_ptr activator,
CORBA::Boolean block = 1)
```

This method instructs the BOA to delay activation of the object implementation associated with the specified `service_name` until a client requests the service. Once a client requests the service, the specified `Activator` object is used to activate the object implementation. If `block` is set to 0, this method blocks the caller until the `exit_impl_ready` method is invoked.

Parameter	Description
<code>service_name</code>	The service name associated with the specified <code>Activator</code> object.
<code>activator</code>	The <code>Activator</code> to be used to activate the object implementation
<code>block</code>	If set to 1, indicates that this method should block the caller. If set to zero, the method does not block. The default behavior is to block.

```
void impl_is_ready(extension::ImplementationDef_ptr impl=NULL)
```

This method notifies the BOA that one or more objects in the server are ready to receive service requests. This method blocks the caller until the `exit_impl_ready` method is invoked. If all objects that the implementation offers were created through C++ instantiation and activated using the `obj_is_ready` method, do not specify the `ImplementationDef_ptr`.

An object implementation may offer only one object and may want to defer the activation of that object until a client request is received. In these cases, the object implementation does not need to first invoke the `obj_is_ready` method. Instead, it may simply invoke this method, passing the `ActivationImplDef` pointer as its single object.

Parameter	Description
<code>impl</code>	This pointer's true type is <code>ActivationImplDef</code> and provides the interface name, object name, path name of the executable and activation policy, along with other parameters.

```
static CORBA::BOA_ptr _nil()
```

This static method returns a NULL BOA pointer that can be used for initialization purposes.

```
void obj_is_ready(CORBA::Object_ptr obj, extension::ImplementationDef_ptr impl_ptr = NULL)
```

This method notifies the BOA that the specified object is ready for use by clients. There are two different ways to use this method:

- Objects that have been created using C++ instantiation should only specify a pointer to the object and let the `ImplementationDef_ptr` default to NULL.
- Objects whose creation is to be deferred until the first client request is received should specify a NULL `Object_ptr` and provide a pointer to an `ActivationImplDef` object that has been initialized.

Parameter	Description
<code>obj</code>	A pointer to the object to be activated.
<code>impl_ptr</code>	A optional pointer to an <code>ActivationImplDef</code> object.

```
static void CORBA::release(CORBA::BOA_ptr boa)
```

This static method releases the specified BOA pointer. Once the object's reference count reaches zero, the object is automatically deleted.

Parameter	Description
<code>boa</code>	A valid BOA pointer.

```
static RegistrationScope scope()
```

This static method returns the registration scope of the BOA. The registration scope of an object can be `SCOPE_GLOBAL` or `SCOPE_LOCAL`. Only objects with a global scope are registered with the osagent.

```
static void scope(RegistrationScope val)
```

This static method changes the registration scope of the BOA to the specified value.

Parameter	Description
<code>val</code>	The scope for this BOA. Must be one of the following values: <code>LOCAL_SCOPE</code> for transient objects. <code>GLOBAL_SCOPE</code> for objects registered with the Smart Agent.

```
void shutdown()
```

This method causes a previous invocation of the `impl_is_ready` method to return. This method cannot be invoked in the context of an active request.

`CORBA::Object_ptr string_to_object(const char * str)`

This method converts a stringified object reference, created with the `object_to_string` method described in “[char *object_to_string\(CORBA::Object_ptr obj\);](#)” on page 27, back into an object reference that may be used to invoke methods on the object.

Parameter	Description
<code>str</code>	The string to be converted back to an object reference.

VisiBroker extensions to CORBA::BOA

`CORBA::ULong connection_max()`

This method returns the maximum number of connections allowed.

`void connection_max(CORBA::ULong max_conn)`

This method is used by servers to set the maximum number of connections allowed.

Parameter	Description
<code>max_conn</code>	The maximum number of connections allowed.

`CORBA::ULong thread_max()`

This method returns the maximum number of threads allowed if the `TPool` thread policy has been selected.

`void thread_max(CORBA::ULong max)`

This method sets the maximum number of threads allowed when the `TPool` thread policy has been selected. If the current number of threads exceeds this number, the extra threads are destroyed one at a time as soon as they are no longer in use until the number of threads is down to `max`.

Parameter	Description
<code>max</code>	The maximum number of threads to be allowed.

`CORBA::ULong thread_stack_size()`

Returns the maximum thread stack size (in bytes) when `TPool` or `TSession` thread policy is selected.

`void thread_stack_size(CORBA::ULong size)`

Sets the maximum thread stack size (in bytes) when `TPool` or `TSession` thread policy is selected.

Parameter	Description
<code>size</code>	The new stack size to be set.

CompletionStatus

```
enum CompletionStatus
```

This enumeration represents how an operation request completed.

IDL Definition

```
enum CompletionStatus {
    COMPLETED_YES;
    COMPLETED_NO;
    COMPLETED_MAYBE;};
```

CompletionStatus members

COMPLETED_YES = 0	Indicates the operation request completed successfully.
COMPLETED_NO = 1	Indicates the operation request was not completed, due to some sort of exception or error.
COMPLETED_MAYBE = 2	Indicates that the operation request may have completed, in spite of an exception or error.

Context

```
class CORBA::Context
```

The `Context` class contains information about a client application's environment that is passed to a server as an implicit parameter during static or dynamic method invocations. It can be used to communicate special information that needs to be associated with a request, but is not part of the method's argument list.

The `Context` class consists of a list of properties, stored as name-value pairs, and provides methods for setting and manipulating those properties. A `Context` contains an `NVList` object and chains the name-value pairs together.

A `Context_var` class is also available and provides simpler memory management semantics.

Include file

Include the `corba.h` file when you use this class.

Context methods

```
const char *context_name() const;
```

This method returns the name used to identify this context. If no name was provided when this object was created, it returns a `NULL` value.

```
void create_child(const char * name, CORBA::Context_out context_ptr);
```

This method creates a child `Context` for this object.

Parameter	Description
<code>name</code>	The name of the new <code>Context</code> object.
<code>context_ptr&</code>	A reference to newly created child <code>Context</code> .

```
void delete_values(const char *name);
```

This method deletes one or more properties from this object.

Parameter	Description
<code>name</code>	The name of the property, or properties, to be deleted. To delete all matching properties, the name may contain a trailing “*” wildcard character. To delete all properties, specify a single asterisk.

```
static CORBA::Context_ptr _duplicate(CORBA::Context_ptr ctx);
```

This method duplicates the specified object.

Parameter	Description
<code>ctx</code>	The object to be duplicated.

```
void get_values(const char *start_scope, CORBA::Flags flag, const char *name,
CORBA::NVList_out NVList_ptr);
```

This method searches the `Context` object hierarchy and retrieves one or more of the name/value pairs specified by the `name` parameter. If the name parameter has a trailing wildcard character (*), then all matching properties and their values are returned. It then creates an `NVList` object and places the name/value pairs in the `NVList`. If the name parameter is an empty string or a NULL string, the `BAD_PARAM` standard system exception is raised. If the name parameter is not found, the `BAD_CONTEXT` standard system exception is raised and no property list is returned.

The `start_scope` parameter specifies the name of the context where the search begins. If the property is not found, the search continues up the `Context` object hierarchy until a match is found, or until there are no more `Context` objects to search. If the `start_scope` parameter is omitted, the search begins with the specified context object.

Parameter	Description
<code>start_scope</code>	The name of the <code>Context</code> object at which to start the search. If set to <code>CORBA::Context::_nil()</code> , the search begins with the current <code>Context</code> . To restrict the search scope can to just the current <code>Context</code> , specify <code>CORBA::CTX_RESTRICT_SCOPE</code> .
<code>flag</code>	An exception is raised if no matching context name is found.
<code>name</code>	The property name to search for. A trailing “*” wildcard character may be used to retrieve all properties that match <code>name</code> .
<code>NVList_ptr</code>	A reference to the list of properties found.

```
static CORBA::Context_ptr _nil();
```

This method returns a NULL `Context_ptr` suitable for initialization purposes.

```
CORBA::Context_ptr parent();
```

This method returns a pointer to the parent `Context`. If there is no parent `Context`, a `NULL` value is returned.

```
static void _release(CORBA::Context_ptr ctx);
```

This static method releases the specified `Context` object. Once the object's reference count reaches zero, the object is automatically deleted.

Parameter	Description
<code>ctx</code>	The object to be released.

```
void set_one_value(const char *name, const CORBA::Any& anAny);
```

This method adds a property to this object using the specified name and value.

Parameter	Description
<code>name</code>	The property's name.
<code>anAny</code>	The property's value.

```
void set_values(CORBA::NVList_ptr _list);
```

This method adds one or more properties to this object, using the name/value pairs specified in the `NVList`. When you create the `NVList` object to be used as an input parameter to this method, you must set the `Flags` field to zero and each `Any` object added to the `NVList` must have its `TypeCode` set to `TC_string`. For more information on the `NVList` class, see [“NVList methods” on page 66](#).

Parameter	Description
<code>_list</code>	A list of name/value pairs to be added to this object.

PortableServer::Current

```
class PortableServer::Current : public CORBA::Current
```

This class provides methods with access to the identity of the object on which the method was called. The `Current` class provides support for servants which implement multiple objects but can be used within the context of POA-dispatched method invocations on any servant.

IDL Definition

```
interface Current : CORBA::Current {
    exception NoContext {};
    OA get_POA() raises (NoContext);
};
```

PortableServer::Current methods

```
PortableServer::POA *get_POA();
```

This method returns a reference to the POA which implements the object in whose context it is called. If this method is called from outside the context of a POA-dispatched method, a `NoContext` exception is raised.

```
PortableServer::ObjectId get_object_id();
```

This method returns the `ObjectId` which identifies the object in whose context it was called. If this method is called from outside the context of a POA-dispatched method, a `NoContext` exception is raised.

Exception

```
class CORBA::Exception
```

The `Exception` class is the base class of the system exception and user exception classes. For more information, see [“SystemException” on page 45](#).

Include file

You should include the `corba.h` file when using this class.

Object

```
class CORBA::Object
```

All ORB objects are derived from the `Object` class, which provides methods for binding clients to objects and manipulating object references as well as querying and setting an object's state. `Object` class methods are implemented by the ORB.

VisiBroker for C++ provides extensions to the CORBA Object specification. These are covered in [“VisiBroker extensions to CORBA::Object” on page 22](#).

Include file

You should include the file `corba.h` when using this class.

CORBA::Object methods

```
void _create_request(CORBA::Context_ptr ctx, const char *operation,
CORBA::NVList_ptr arg_list, CORBA::NamedValue_ptr result, CORBA::Request_out
request, CORBA::Flags req_flags);
```

This method creates a `Request` for an object implementation that is suitable for invocation with the Dynamic Invocation Interface.

Parameter	Description
<code>ctx</code>	The Context associated with this request. For more information, see “CompletionStatus” on page 16 .
<code>operation</code>	The name of the operation to be performed on the object implementation.
<code>arg_list</code>	A list of arguments to pass to the object implementation. See the Dynamic interfaces and classes, “NVList methods” on page 66 for more information.
<code>result</code>	The result of the operation. See the Dynamic interfaces and classes, “NamedValue methods” on page 65 for more information.
<code>request</code>	A pointer to the <code>Request</code> that is created. For more information, see the Dynamic interfaces and classes, “Request methods” on page 69 .
<code>req_flags</code>	The <code>OUT_LIST_MEMORY</code> and <code>IN_COPY_VALUE</code> flags can be set as flags in the <code>req_flags</code> parameter, but they are meaningless and thus ignored because argument insertion and extraction are done via the <code>Any</code> type.

```
void _create_request(CORBA::Context_ptr ctx, const char *operation,
CORBA::NVList_ptr arg_list, CORBA::NamedValue_ptr result, CORBA::ExceptionList_ptr
eList, CORBA::ContextList_ptr ctxList, CORBA::Request_out request, CORBA::Flags
req_flags);
```

This method creates a `Request` for an object implementation that is suitable for invocation with the Dynamic Invocation Interface.

Parameter	Description
<code>ctx</code>	The Context associated with this request. For more information, see “CompletionStatus” on page 16 .
<code>operation</code>	The name of the operation to be performed on the object implementation.
<code>arg_list</code>	A list of arguments to pass to the object implementation. See the Dynamic interfaces and classes, “NVList methods” on page 66 for more information.
<code>result</code>	The result of the operation. See the Dynamic interfaces and classes, “NamedValue methods” on page 65 for more information.
<code>eList</code>	A list of exceptions for this request.
<code>ctxList</code>	A list of <code>Context</code> objects for this request.
<code>request</code>	A pointer to the <code>Request</code> that is created. See the Dynamic interfaces and classes, “Request methods” on page 69 for more information.
<code>req_flags</code>	This flag must be set to <code>OUT_LIST_MEMORY</code> if one or more of the <code>NamedValue</code> items in <code>arg_list</code> are output arguments.

```
static CORBA::Object_ptr _duplicate(CORBA::Object_ptr obj);
```

This static method duplicates the specified `Object_ptr` and returns a pointer to the object. The object's reference count is increased by one.

Parameter	Description
<code>obj</code>	The object pointer to be duplicated.

```
CORBA::InterfaceDef_ptr _get_interface();
```

This method returns a pointer to this object's interface definition. See the Interface repository interfaces and classes, [“InterfaceDef methods” on page 97](#) for more information.

```
CORBA::ULong _hash(CORBA::ULong maximum);
```

This method returns a hash value for this object. This value does change for the lifetime of this object, however the value is not necessarily unique. If two objects return different hash values, then they are not identical. The upper bound of the hash value may be specified. The lower bound is 0 (zero).

Parameter	Description
maximum	The upper bound of the hash value returned.

```
CORBA::Boolean _is_a(const char *logical_type_id);
```

This method returns `TRUE` if this object implements the interface associated with the repository id. Otherwise, it returns `FALSE`.

Parameter	Description
logical_type_id	The repository identifier to check.

```
CORBA::Boolean _is_equivalent(CORBA::Object_ptr other_object);
```

This method returns `TRUE` if the specified object pointer and this object point to the same object implementation. Otherwise, it returns `FALSE`.

Parameter	Description
other_object	Pointer to an object that is to be compared to this object.

```
static CORBA::Object_ptr _nil();
```

This static method returns a `NULL` pointer suitable for initialization purposes.

```
CORBA::Boolean _non_existent();
```

This method returns `TRUE` if the object represented by this object reference no longer exists.

```
CORBA::Request_ptr _request(const char* operation);
```

This method creates a `Request` suitable for invoking methods on this object. A pointer to the `Request` object is returned. See the Dynamic interfaces and classes, [“Request methods” on page 69](#) for more information.

Parameter	Description
operation	The name of the object method to be invoked.

```
CORBA::Object_ptr _resolve_reference(const char* id);
```

Your client application can invoke this method on an object reference to resolve the server-side interface with the specified service identifier. This method causes the `ORB::_resolve_initial_references` method to be invoked on the server-side to resolve the specified service. This method returns an object reference which your client can narrow to the appropriate server type.

This method is typically used by client applications that wish to manage a server's attributes.

Parameter	Description
id	The name of the interface to be resolved on the server-side.

VisiBroker extensions to CORBA::Object

```
CORBA::BindOptions* _bind_options();
```

This method returns a pointer to the bind options that used for this object only. For more information, see [“BindOptions” on page 10](#).

```
void _bind_options(const CORBA::BindOptions& opt);
```

This method sets the bind options for this object only. The options that are set remain in effect for the lifetime of the proxy object. Any changes to time-out values will apply to all subsequent send and receive operations as well as any re-bind operations. For more information, see [“BindOptions” on page 10](#).

Parameter	Description
opt	The new bind options for this object.

```
static CORBA::Object_ptr _bind_to_object(const char *rep_id, const char
*object_name=NULL, const char *host_name=NULL, const CORBA::BindOptions
*options=NULL, CORBA::ORB_ptr orb=NULL);
```

This method attempts to bind to the object with the specified `repository_id` and `object_name` on the specified host using the specified `BindOptions` and ORB.

Parameter	Description
rep_id	The repository ID of the desired object.
object_name	The name of the desired object.
host_name	The name of the desired host where the object implementation is executing.
options	The bind options for this connection. See “struct BindOptions” on page 10 for more information.
orb	The ORB to use.

```
CORBA::BOA _boa() const;
```

This method returns a pointer to the Basic Object Adaptor with which this object is registered.

Note This method is deprecated in VisiBroker 4.0.

```
static CORBA::Object_ptr _clone(CORBA::Object_ptr obj, CORBA::Boolean
reset_connection = 1UL);
```

This method clones the specified object reference.

Parameter	Description
obj	The object reference to be cloned.
reset_connection	This parameter is not used.


```
static const CORBA::BindOptions * _default_bind_options();
```

This method returns a pointer to the global, per client process BindOptions. For more information, see [“BindOptions” on page 10](#).

```
static void _default_bind_options(const CORBA::BindOptions& bindOptions);
```

This method sets the bind options to be used by default for all `_bind` invocations that do not specify their own bind options. For more information, see [“BindOptions” on page 10](#).

```
static const CORBA::TypeInfo *_desc();
```

Returns type information for this object.

```
const char *_interface_name() const;
```

This method returns this object's interface name.

```
CORBA::Boolean _is_bound() const;
```

If the client process has established a connection to an object implementation (is bound), this method returns 1. If the object is not bound, this method returns 0 zero.

```
CORBA::Boolean _is_local() const;
```

This method returns `TRUE` if the object implementation resides within the same process or address space as the client application.

```
CORBA::Boolean _is_persistent() const;
```

This method returns `TRUE` if this object is a persistent object. A `FALSE` value returned is not an authoritative answer that the object is not persistent.

```
CORBA::Boolean _is_remote() const;
```

This method returns `TRUE` if the object implementation resides in a different process or address space than the client application. The client and object implementation may or may not reside on the same host.

```
const char *_object_name() const;
```

This method returns the object name associated with this object.

```
CORBA::Long _ref_count() const;
```

Returns the reference count for this object.

```
void _release();
```

Decrements this object's reference count and releases the object if the reference count has reached 0.

```
const char *_repository_id() const;
```

This method returns this object's repository identifier.

ORB

```
class CORBA::ORB
```

The `ORB` class provides an interface to the Object Request Broker. It provides methods to the client object, independent of the particular `Object` or `Object Adaptor`.

Borland VisiBroker provides extensions to the CORBA ORB that are discussed in [“VisiBroker extensions to CORBA::ORB” on page 29](#). These methods are provided for the management of connections, threads, and the activation of services.

Include file

You should include the file `corba.h` when using this class.

CORBA::ORB methods

```
CORBA::Boolean work_pending();
```

This method returns true if the ORB has any work waiting to be processed.

```
static CORBA::TypeCode_ptr create_alias_tc(const char *repository_id, const char
*type_name, CORBA::TypeCode_ptr original_type);
```

This static method dynamically creates a `TypeCode` for the alias with the specified type and name.

Parameter	Description
<code>repository_id</code>	The identifier generated by the IDL compiler or constructed dynamically.
<code>type_name</code>	The name of the alias's type.
<code>original_type</code>	The type of the original for which this alias is being created.

```
static CORBA::TypeCode_ptr create_array_tc(CORBA::Ulong length, TypeCode_ptr
element_type);
```

This static method dynamically creates a `TypeCode` for an array.

Parameter	Description
<code>length</code>	The maximum number of array elements.
<code>element_type</code>	The type of elements stored in this array.

```
static CORBA::TypeCode_ptr create_enum_tc(const char *repository_id, const char
*type_name, const CORBA::EnummemberSeq& members);
```

This static method dynamically creates a `TypeCode` for an enumeration with the specified type and members.

Parameter	Description
<code>repository_id</code>	The identifier generated by the IDL compiler or constructed dynamically.
<code>type_name</code>	The name of the enumeration's type.
<code>members</code>	A list of values for the enumeration's members.

```
void create_environment(CORBA::Environment_out env);
```

This method creates an `Environment` object.

Parameter	Description
env	The reference that will be set to point to the newly created <code>Environment</code> .

```
static CORBA::TypeCode_ptr create_exception_tc(const char *repository_id, const char *type_name, const CORBA::StructMemberSeq& members);
```

This static method dynamically creates a `TypeCode` for an exception with the specified type and members.

Parameter	Description
repository_id	The identifier generated by the IDL compiler or constructed dynamically.
type_name	The name of the structure's type.
members	A list of values for the structure members.

```
static CORBA::TypeCode_ptr create_interface_tc(const char *repository_id, const char *type_name);
```

This static method dynamically creates a `TypeCode` for the interface with the specified type.

Parameter	Description
repository_id	The identifier generated by the IDL compiler or constructed dynamically.
type_name	The name of the interface's type.

```
void create_list(CORBA::Long num, CORBA::NVList_out nvList);
```

This method creates an `NVList` with the specified number of elements and returns a reference to the list.

Parameter	Description
num	The number of elements in the list.
nvlist	Initialized to point to the newly created list.

```
void create_named_value(CORBA::NamedValue_out value);
```

This method creates a `NamedValue` object.

```
void create_operation_list(CORBA::OperationDef_ptr opDefPtr, CORBA::NVList_out nvList);
```

This method creates an argument list for the specified `OperationDef` object.

```
static CORBA::TypeCode_ptr create_recursive_sequence_tc(CORBA::Ulong bound, CORBA::Ulong offset);
```

This static method dynamically creates a `TypeCode` for a recursive sequence. The result of this method can be used to create other types. The `offset` parameter determines which enclosing `TypeCode` describes the elements of this sequence.

Parameter	Description
bound	The maximum number of sequence elements.
offset	Position within the buffer where the type code for the current element was previously generated.

```
static CORBA::TypeCode_ptr create_sequence_tc(CORBA::Ulong bound,
CORBA::TypeCode_ptr element_type);
```

This static method dynamically creates a `TypeCode` for a sequence.

Parameter	Description
<code>bound</code>	The maximum number of sequence elements.
<code>element_type</code>	The type of elements stored in this sequence.

```
static CORBA::TypeCode_ptr create_string_tc(CORBA::Ulong bound);
```

This static method dynamically creates a `TypeCode` for a string.

Parameter	Description
<code>bound</code>	The maximum length of the string.

```
static CORBA::TypeCode_ptr create_struct_tc(const char *repository_id, const char
*type_name, const ORBA::StructMemberSeq& members);
```

This static method dynamically creates a `TypeCode` for the structure with the specified type and members.

Parameter	Description
<code>repository_id</code>	The identifier generated by the IDL compiler or constructed dynamically.
<code>type_name</code>	The name of the structure's type.
<code>members</code>	A list of values for the structure members.

```
static CORBA::TypeCode_ptr create_union_tc(const char *repository_id, const char
*type_name, CORBA::TypeCode_ptr discriminator_type, const CORBA::UnionMemberSeq&
members);
```

This static method dynamically creates a `TypeCode` for a union with the specified type, discriminator and members.

Parameter	Description
<code>repository_id</code>	The identifier generated by the IDL compiler or constructed dynamically.
<code>type_name</code>	The name of the union's type.
<code>discriminator_type</code>	The discriminating type for the union.
<code>members</code>	A list of values for the union members.

```
CORBA::Status get_default_context(CORBA::Context_ptr& LcontextPtr);
```

This method returns the default per-process `ContextL` maintained by `VisiBroker`. The default `Context` is often used in constructing DII requests. For more information, see [“Context” on page 16](#).

Parameter	Description
<code>contextPtr&</code>	The property's value.

```
CORBA::Status get_next_response(CORBA::Request_out* & req);
```

This method blocks waiting for the response associated with a deferred request. You can use the `ORB::poll_next_response` method to determine if there is a response waiting to be received before you call this method.

Parameter	Description
<code>req</code>	Set to point to the request that has been received.

```
ObjectIDList *list_initial_services();
```

This method returns a list of the names of any object services that are available to your application. These services may include the Location Service, Interface Repository, Name Service, or Event Service. You can use any of the returned names with the `ORB::resolve_initial_references` method to obtain the top-level object for that service.

```
char *object_to_string(CORBA::Object_ptr obj);
```

This method converts the specified object reference to a string, a process referred to as “stringification” in the CORBA specification. Object references that have been converted to strings can be stored in files, for example. This is an ORB method because different ORB implementations may have different conventions for representing object references as strings.

```
CORBA::BOA_ptr ORB::BOA_init(int& argc, char *const *argv, const char *boa_identifier = (char *)NULL);
```

This ORB method returns a handle to the BOA and specifies optional networking parameters. The `argc` and `argv` parameters are the same parameters passed to the object implementation process when it is started.

Note This method is deprecated since VisiBroker 4.0.

Parameter	Description
<code>argc</code>	The number of arguments passed.
<code>argv</code>	An array of char pointers to the arguments. All but two of the arguments take the form of a keyword and a value. This method ignores any keywords that it does not recognize.
<code>boa_identifier</code>	Identifies the type of BOA to be used. Use <code>TPool</code> if multiple thread support is desired. Use <code>TSingle</code> if the implementation does not use threads.

```
static CORBA::ORB_ptr ORB_init(int& argc, char *const *argv, const char *orb_id = NULL);
```

This method initializes the ORB and is used by both clients and object implementations. It returns a pointer to the ORB that can be used to invoke ORB methods. The `argc` and `argv` parameters passed to the application's main function can be passed directly to this method. Arguments accepted by this method take the form of name-value pairs so that they can be distinguished from other command line arguments.

Parameter	Description
<code>argc</code>	The number of arguments passed.
<code>argv</code>	An array of char pointers to the arguments. All but two of the arguments take the form of a keyword and a value. This method ignores any keywords that it does not recognize.
<code>boa_identifier</code>	Identifies the type of ORB to be used. The default is <code>IIOP</code> .

```
void perform_work();
```

This method instructs the ORB to perform some work.

```
CORBA::Boolean poll_next_response();
```

This method returns `TRUE` if a response to a deferred request was received, otherwise `FALSE` is returned. This call does not block.

```
CORBA::Object_ptr resolve_initial_references(const char * identifier);
```

This method resolves one of the names returned by the `ORB::list_initial_services` method, described in “[ObjectIDList *list_initial_services\(\)](#),” on page 27, to its corresponding implementation object. The resolved object which is returned can then be narrowed to the appropriate server type. If the specified service cannot be found, an `InvalidName` exception is raised.

Parameter	Description
<code>identifier</code>	The name of the service whose top-level object is to be returned. The identifier is not the name of the object to be returned.

```
void send_multiple_requests_deferred(const CORBA::RequestSeq& req);
```

This method sends all the client requests in the specified sequence as deferred requests. The ORB will not wait for any responses from the object implementation. The client application is responsible for retrieving the responses to each request using the `ORB::get_next_response` method.

Parameter	Description
<code>req</code>	A sequence of deferred requests to be sent.

```
void send_multiple_requests_oneway(const CORBA::RequestSeq& req);
```

This method sends all the client requests in the specified sequence as one-way requests. The ORB does not wait for a response from any of the requests because one-way requests do not generate responses from the object implementation.

Parameter	Description
<code>req</code>	A sequence of one-way requests to be sent.

```
CORBA::Object_ptr string_to_object(const char *str);
```

This method converts a string representing an object into an object pointer. The string must have been created using the `ORB::object_to_string` method.

Parameter	Description
<code>str</code>	A pointer to a string representing an object.

```
static CORBA::ORB_ptr _duplicate(CORBA::ORB_ptr ptr);
```

This static method duplicates the specified ORB pointer and returns a pointer to the duplicated ORB.

Parameter	Description
<code>ptr</code>	The ORB pointer to be duplicated.

```
static CORBA::ORB_ptr _nil();
```

This static method returns a NULL ORB pointer suitable for initialization purposes.

```
void run();
```

This method causes the ORB to start processing work. This ORB receives requests and dispatches them. This call blocks this process until the ORB is shut down.

```
static void shutdown(CORBA::Boolean wait_for_completion=0);
```

This method causes a previous invocation of the `impl_is_ready` method to return. All object adapters are shut down and associated memory is freed. If the `wait_for_completion` parameter is `TRUE`, this operation blocks until the shutdown is complete. If an application does this in a thread that is currently servicing an invocation, the `BAD_INV_ORDER` system exception is raised. If the `wait_for_completion` parameter is `FALSE`, then shutdown may not have completed upon return.

Parameter	Description
<code>wait_for_completion</code>	Specifies whether shutdown will wait for completion or not

```
static void destroy();
```

This operation destroys the ORB so that its resources can be reclaimed by the application. If `destroy` is called on an ORB that has not been shut down, it will start the shut down process and block until the ORB has shut down before it destroys the ORB. The behavior is similar to that achieved by calling `shutdown` with the `wait_for_completion` parameter set to `TRUE`. If an application calls `destroy` in a thread that is currently servicing an invocation, the `BAD_INV_ORDER` system exception is raised.

VisiBroker extensions to CORBA::ORB

```
CORBA::Object_ptr bind(const char *rep_id, const char *object_name = (const char*)NULL, const char *host_name = (const char*)NULL, CORBA::BindOptions *opt = (CORBA::BindOptions*)NULL);
```

This method allows you obtain a generic object reference to an object by specifying the repository id of the object and optionally, its object name and host name where it is implemented.

Parameter	Description
<code>rep_id</code>	The identifier generated by the IDL compiler or constructed dynamically for the object.
<code>object_name</code>	The name of the object. This is an optional parameter.
<code>host_name</code>	The host name where the object implementation is located. This may be specified as an IP address or as a fully qualified host name. This is an optional parameter.
<code>opt</code>	Any bind options for the object. This is an optional parameter. Bind options are described in “BindOptions” on page 10 .

```
CORBA::ULong connection_count();
```

This method is used by client applications to return the current number of active connections.

```
void connection_max(CORBA::ULong max_conn)
```

Client applications use this method to set the maximum number of connections allowed.

Parameter	Description
max_conn	The maximum number of connections to be allowed.

```
CORBA::ULong connection_max()
```

Client applications use this method to obtain the maximum number of connections allowed.

```
static CORBA::TypeCode_ptr create_wstring_tc(CORBA::ULong bound);
```

This static method dynamically creates a `TypeCode` for a Unicode string.

Parameter	Description
bound	The maximum length of the string.

PortableServer::POA

```
class PortableServer::POA
```

Objects of the POA class manage the implementations of a collection of objects. The POA supports a namespace for these objects which are identified by Object Ids. A POA also provides a namespace for other POAs in that a POA must be created as a child of an existing POA, which then forms a hierarchy starting with the root POA.

A POA object must not be exported to other processes or be stringified. A `MARSHAL` exception is raised if this is attempted.

PortableServer::POA methods

```
PortableServer::ObjectId* activate_object(PortableServer::Servant _p_servant);
```

This method generates an object id and returns it. The object id and the specified `_p_servant` are entered into the Active Object Map. If the `UNIQUE_ID` policy is present with the POA and the specified `_p_servant` is already in the Active Object Map, then a `ServantAlreadyActive` exception is raised.

This method requires that the `SYSTEM_ID` and `RETAIN` policies be present with the POA; otherwise, a `WrongPolicy` exception is raised.

Parameter	Description
_p_servant	The Servant to be entered into the Active Object Map.

```
void activate_object_with_id(const PortableServer::ObjectId& _oid,
PortableServer::Servant _p_servant);
```

This method attempts to activate the specified `_oid` and to associate it with the specified `_p_servant` in the Active Object Map. If the `_oid` already has a servant bound to it in the Active Object Map, then an `ObjectAlreadyActive` exception is raised. If the POA has the `UNIQUE_ID` policy present and the `_p_servant` is already in the Active Object map, then a `ServantAlreadyActive` exception is raised.

If the POA has the `SYSTEM_ID` policy present and it detects that the `_oid` was not generated by the system or for the POA, then this method raises a `BAD_PARAM` system exception.

This method requires that the `RETAIN` policy be present with the POA; otherwise, a `WrongPolicy` exception is raised.

Parameter	Description
<code>_oid</code>	The Objectid of the object to be activated.
<code>_p_servant</code>	The Servant to be entered into the Active Object Map.

```
PortableServer::ImplicitActivationPolicy_ptr
create_implicit_activation_policy(PortableServer::ImplicitActivationPolicyValue
_value);
```

This method returns a pointer to an `ImplicitActivationPolicy` object with the specified `_value`. The application is responsible for calling the inherited `destroy` method on the `Policy` object after the `Policy` object is no longer needed.

If no `ImplicitActivationPolicy` is specified at POA creation, then the default is `NO_IMPLICIT_ACTIVATION`.

Parameter	Description
<code>_value</code>	If set to <code>IMPLICIT_ACTIVATION</code> , the POA implicitly activates servants: also requires <code>SYSTEM_ID</code> and <code>RETAIN</code> policies. If set to <code>NO_IMPLICIT_ACTIVATION</code> , the POA will not implicitly activate servants.

```
CORBA::Object_ptr create_reference(const char* _intf);
```

This method creates and returns an object reference that encapsulates a POA-generated `ObjectId` and the specified `_intf` values. The `_intf`, which may be null, becomes the `type_id` of the generated object reference. This method does not cause an activation to take place. Undefined behavior results if the `_intf` value does not identify the most derived interface of the object or one of its base interfaces. The `ObjectId` may be obtained by invoking the `POA::reference_to_id` method on the returned `Object`.

This method requires that the `RETAIN` policy be present with the POA; otherwise, a `WrongPolicy` exception is raised.

Parameter	Description
<code>_intf</code>	The repository interface id of the class of the object to be created.

```
CORBA::Object_ptr create_reference_with_id (const PortableServer::ObjectId&
_oid, const char* _intf);
```

This method creates and returns an object reference that encapsulates the specified `_oid` and `_intf` values. The `_intf`, which may be a null string, becomes the `type_id` of the generated object reference. An `_intf` value that does not identify the most derived interface of the object or one of its base interfaces will result in undefined behavior. This method does not cause an activation to take place. The returned object reference may be passed to clients, so that subsequent requests on those references will cause the object to be activated if necessary, or the default servant used, depending on the applicable policies.

If the POA has the `SYSTEM_ID` policy present and it detects the `ObjectId` value was not generated by the system or for the POA, this method may raise a `BAD_PARAM` system exception.

Parameter	Description
<code>_oid</code>	The object id for which a reference is to be created.
<code>_intf</code>	The repository interface id of the class of the object to be created.

```
PortableServer::IdAssignmentPolicy_ptr create_id_assignment_policy
(PortableServer::IdAssignmentPolicyValue _value);
```

This method returns a pointer to an `IdAssignmentPolicy` object with the specified `_value`. The application is responsible for calling the inherited destroy method on the `Policy` object after it is no longer needed.

If no `IdAssignmentPolicy` is specified at POA creation, then the default is `SYSTEM_ID`.

Parameter	Description
<code>_value</code>	If set to <code>USER_ID</code> , then objects created by the POA are assigned object ids only by the application. If set to <code>SYSTEM_ID</code> , then objects created by the POA are assigned object ids only by the POA.

```
PortableServer::IdUniquenessPolicy_ptr create_id_uniqueness_policy
(PortableServer::IdUniquenessPolicyValue _value);
```

This method returns a pointer to an `IdUniquenessPolicy` object with the specified `_value`. The application is responsible for calling the inherited destroy method on the `Policy` object after it is no longer needed.

If no `IdUniquenessPolicy` is specified at POA creation, then the default is `UNIQUE_ID`.

Parameter	Description
<code>_value</code>	If set to <code>UNIQUE_ID</code> , then servants which are activated with the POA support exactly one object id. If set to <code>MULTIPLE_ID</code> , then a servant which is activated with the POA may support one or more object ids.

```
PortableServer::LifespanPolicy_ptr create_lifespan_policy
(PortableServer::LifespanPolicyValue _value);
```

This method returns a pointer to a `LifespanPolicy` object with the specified `_value`. The application is responsible for calling the inherited destroy method on the `Policy` object after it is no longer needed.

If no `LifespanPolicy` is specified at POA creation, then the default is `TRANSIENT`.

Parameter	Description
<code>_value</code>	If set to <code>TRANSIENT</code> , then objects implemented in the POA cannot outlive the POA instance in which they were first created. Once a transient POA is deactivated, the use of any object references generated from it results in an <code>OBJECT_NOT_EXIST</code> exception being raised. If set to <code>PERSISTENT</code> , then the objects implemented in the POA can outlive any process in which they are first created.

```
PortableServer::POA_ptr create_POA(const char* _adapter_name,
PortableServer::POAManager_ptr _a_POAManager, const CORBA::PolicyList& _policies);
```

This method creates a new POA with the specified `_adapter_name`. The new POA is a child of the specified `_a_POAManager`. If a child POA with the same name already exists for the parent POA, a `PortableServer::AdapterAlreadyExists` exception is raised.

The specified `_policies` are associated with the new POA and are used to control its behavior.

Parameter	Description
<code>_adapter_name</code>	The name which specifies the new POA.
<code>_a_POAManager</code>	The parent POA object of the new POA.
<code>_policies</code>	A list of policies which are to apply to the new POA. The policy objects are effectively copied before this operation returns.

```
PortableServer::RequestProcessingPolicy_ptr create_request_processing_policy
(PortableServer::RequestProcessingPolicyValue _value);
```

This method returns a pointer to a `RequestProcessingPolicy` object with the specified `_value`. The application is responsible for calling the inherited destroy method on the `Policy` object after it is no longer needed.

If no `RequestProcessingPolicy` is specified at POA creation, then the default is `USE_ACTIVE_OBJECT_MAP_ONLY`.

Parameter	Description
<code>_value</code>	If set to <code>USE_ACTIVE_OBJECT_MAP_ONLY</code> and the object id is not found in the Active Object Map, then an <code>OBJECT_NOT_EXIST</code> exception is returned to the client. (The <code>RETAIN</code> policy is also required.) If set to <code>USE_DEFAULT_SERVANT</code> and the object id is not found in the Active Object Map or the <code>NON_RETAIN</code> policy is present, and a default servant has been registered with the POA using the <code>set_servant</code> method, then the request is dispatched to the default servant. If no default servant has been registered, then an <code>OBJ_ADAPTER</code> exception is returned to the client. (The <code>MULTIPLE_ID</code> policy is also required.) If set to <code>USE_SERVANT_MANAGER</code> and the object id is not found in the Active Object Map or the <code>NON_RETAIN</code> policy is present, and a servant manager has been registered with the POA using the <code>set_servant_manager</code> method, then the servant manager is given the opportunity to locate a servant or raise an exception. If no servant manager has been registered, then an <code>OBJ_ADAPTER</code> is returned to the client.

```
PortableServer::ServantRetentionPolicy_ptr create_servant_retention_policy
(PortableServer::ServantRetentionPolicyValue _value);
```

This method returns a pointer to a `ServantRetentionPolicy` object with the specified `_value`. The application is responsible for calling the inherited destroy method on the `Policy` object after it is no longer needed.

If no `ServantRetentionPolicy` is specified at POA creation, then the default is `RETAIN`.

Parameter	Description
<code>_value</code>	If set to <code>RETAIN</code> , then the POA will retain active servants in its Active Object Map. If set to <code>NON_RETAIN</code> , then servants are not retained by the POA.

```
PortableServer::ThreadPolicy_ptr
create_thread_policy(PortableServer::ThreadPolicyValue _value);
```

This method returns a pointer to a `ThreadPolicy` object with the specified `_value`. The application is responsible for calling the inherited destroy method on the `Policy` object after it is no longer needed.

If no `ThreadPolicy` is specified at POA creation, then the default is `ORB_CTRL_MODEL`.

Parameter	Description
<code>_value</code>	If set to <code>ORB_CTRL_MODEL</code> , the ORB is responsible for assigning requests for an ORB-controlled POA to threads. In a multi-threaded environment, concurrent requests may be delivered using multiple threads. If set to <code>SINGLE_THREAD_MODEL</code> , then requests to the POA are processed sequentially. In a multi-threaded environment, all upcalls made by the POA to servants and servant managers are made in a manner that is safe for code that is multi-thread unaware.

```
void deactivate_object(const PortableServer::ObjectId& _oid);
```

This method causes the specified `_oid` to be deactivated. An `ObjectId` which has been deactivated continues to process requests until there are no more active requests for that `ObjectId`. An `ObjectId` is removed from the Active Object Map when all requests executing for that `ObjectId` have completed.

If a `ServantManager` is associated with the POA, then the `ServantActivator::etherealize` method is invoked with the `ObjectId` and the associated servant after the `ObjectId` has been removed from the Active Object map. Reactivation for the `ObjectId` blocks until etherealization, if necessary, has completed. However, the method does not wait for requests or etherealization to complete and always returns immediately after deactivating the specified `_oid`.

This method requires that the `RETAIN` policy be present with the POA; otherwise, a `WrongPolicy` exception is raised.

Parameter	Description
<code>_oid</code>	The <code>ObjectId</code> of the object to be deactivated.

```
void destroy(CORBA::Boolean _etherealize_objects, CORBA::Boolean
_wait_for_completion);
```

This method destroys this POA object and all of its descendant POAs. First the children are destroyed and finally the current container POA. If desired, later, a POA with that same name in the same process can be created.

Parameter	Description
<code>_etherealize_objects</code>	If <code>TRUE</code> , the POA has the <code>RETAIN</code> policy, and a servant manager has registered with the POA, then the <code>etherealize</code> method is called on each active object in the Active Object Map. The apparent destruction of the POA occurs before the <code>etherealize</code> method is called, and thus any <code>etherealize</code> method which attempts to invoke methods on the POA raises an <code>OBJECT_NOT_EXIST</code> exception.
<code>_wait_for_completion</code>	If <code>TRUE</code> and the current thread is not in an invocation context dispatched from some POA belonging to the same ORB as this POA, the <code>destroy</code> method only returns after all active requests and all invocations of <code>etherealize</code> have completed. If <code>FALSE</code> and the current thread is in an invocation context dispatched from some POA belonging to the same ORB as this POA, the <code>BAD_INV_ORDER</code> exception is raised and POA destruction does not occur.

```
PortableServer::POA_ptr find_POA(const char* _adapter_name, CORBA::Boolean
_activate_it);
```

If the POA object on which this method is called is the parent of the POA with the specified `_adapter_name`, the child POA is returned.

Parameter	Description
<code>_adapter_name</code>	The name of the AdapterActivator associated with the POA.
<code>_activate_it</code>	If set to <code>TRUE</code> and no child POA of the POA specified by <code>_adapter_name</code> exists, then the POA's AdapterActivator, if not <code>null</code> , is invoked. If it successfully activates the child POA, then that POA is returned. Otherwise an <code>AdapterNonExistent</code> exception is raised.

```
PortableServer::Servant get_servant();
```

This method returns the default `Servant` associated with the POA. If no `Servant` has been associated with the POA, then a `NoServant` exception is raised.

This method requires that the `USE_DEFAULT_SERVANT` policy be present with the POA; otherwise, a `WrongPolicy` exception is raised.

```
PortableServer::ServantManager_ptr get_servant_manager();
```

This method returns a pointer to the `ServantManager` object associated with the POA. The result is `null` if no `ServantManager` is associated with the POA.

This method requires that the `USE_SERVANT_MANAGER` policy be present with the POA; otherwise, a `WrongPolicy` exception is raised.

```
CORBA::Object_ptr id_to_reference(PortableServer::ObjectId& _oid);
```

This method returns an object reference if the specified `_oid` value is currently active. If the `_oid` is not active, then an `ObjectNotActive` exception is raised.

This method requires that the `RETAIN` policy be present with the POA; otherwise, a `WrongPolicy` exception is raised.

Parameter	Description
<code>_oid</code>	The <code>ObjectId</code> of the object for which a reference is to be returned.

```
PortableServer::Servant id_to_servant(PortableServer::ObjectId& _oid);
```

This method has three behaviors:

- If the POA has the `RETAIN` policy present and the specified `_oid` is in the Active Object Map, then it returns the servant associated with that object in the Active Object Map.
- If the POA has the `USE_DEFAULT_SERVANT` policy present and a default servant has been registered with the POA, it returns the default servant.
- Otherwise, an `ObjectNotActive` exception is raised.

This method requires that the `USE_DEFAULT_SERVANT` policy be present with the POA; if neither policy is present, a `WrongPolicy` exception is raised.

Parameter	Description
<code>_oid</code>	The <code>ObjectId</code> of the object for which a servant is to be returned.

```
PortableServer::Servant reference_to_servant(CORBA::Object_ptr _reference);
```

This method has three behaviors:

- If the POA has the `RETAIN` policy and the specified `_reference` is present in the Active Object Map, then it returns the servant associated with that object in the Active Object Map.
- If the POA has the `USE_DEFAULT_SERVANT` policy present and a default servant has been registered with the POA, then it returns the default servant.
- Otherwise, it raises an `ObjectNotActive` exception.

This method requires that the `RETAIN` or `USE_DEFAULT_SERVANT` policies be present; otherwise, a `WrongPolicy` exception is raised.

Parameter	Description
<code>_reference</code>	The object for which a servant is to be returned.

```
PortableServer::ObjectId* reference_to_id(CORBA::Object_ptr _reference);
```

This method returns the `ObjectId` value encapsulated by the specified `_reference`. The invocation is valid only if the `_reference` was created by the POA on which the method is called. If the `_reference` was not created by the POA, a `WrongAdapter` exception is raised. The object denoted by the `_reference` parameter does not have to be active for this method to succeed.

Though the IDL specifies that a `WrongPolicy` exception may be raised by this method, it is simply declared for possible future extension.

Parameter	Description
<code>_reference</code>	The object for which an <code>ObjectId</code> is to be returned.

```
PortableServer::ObjectId* servant_to_id(PortableServer::Servant _p_servant);
```

This method has four possible behaviors:

- If the POA has the `UNIQUE_ID` policy present and the specified `_p_servant` is active, then the `ObjectId` associated with the `_p_servant` is returned.
- If the POA has the `IMPLICIT_ACTIVATION` policy present and either the POA has the `MULTIPLE_ID` policy present or the specified `_p_servant` is not active, then the `_p_servant` is activated using the POA-generated `ObjectId` and the repository interface id associated with the `_p_servant`, and that `ObjectId` is returned.
- If the POA has the `USE_DEFAULT_SERVANT` policy present, the specified `_p_servant` is the default servant, then the `ObjectId` associated with the current invocation is returned.
- Otherwise, a `ServantNotActive` exception is raised.

This method requires that the `USE_DEFAULT_SERVANT` policy or a combination of the `RETAIN` policy and either the `UNIQUE_ID` or `IMPLICIT_ACTIVATION` policies be present; otherwise, a `WrongPolicy` exception is raised.

Parameter	Description
<code>_p_servant</code>	The Servant for which the <code>ObjectId</code> to be returned is desired.

```
CORBA::Object_ptr servant_to_reference(PortableServer::Servant _p_servant);
```

This method has the following possible behaviors:

- If the POA has both the `RETAIN` and the `UNIQUE_ID` policies present and the specified `_p_servant` is active, then an object reference encapsulating the information used to activate the servant is returned.
- If the POA has both the `RETAIN` and the `IMPLICIT_ACTIVATION` policies present and either the POA has the `MULTIPLE_ID` policy or the specified `_p_servant` is not active, then the `_p_servant` is activated using a POA-generated `ObjectId` and repository interface id associated with the `_p_servant`, and a corresponding object reference is returned.
- If this method was invoked in the context of executing a request on the specified `_p_servant`, the reference associated with the current invocation is returned.
- Otherwise, a `ServantNotActive` exception is raised.

This method requires the presence of the `RETAIN` policy and either the `UNIQUE_ID` or `IMPLICIT_ACTIVATION` policies if invoked outside the context of a method dispatched by the POA. If this method is not invoked in the context of executing a request on the specified `_p_servant` and one of these policies is not present, then a `WrongPolicy` exception is raised.

Parameter	Description
<code>_p_servant</code>	The <code>Servant</code> for which a reference is to be returned.

```
void set_servant(PortableServer::Servant _p_servant);
```

This method sets the default `Servant` associated with the POA. The specified `Servant` will be used for all requests for which no servant is found in the Active Object Map.

This method requires that the `USE_DEFAULT_SERVANT` policy be present with the POA; otherwise, a `WrongPolicy` exception is raised.

Parameter	Description
<code>_p_servant</code>	The <code>Servant</code> to be used as the default associated with the POA.

```
void set_servant_manager(PortableServer::ServantManager_ptr _imagr);
```

This method sets the default `ServantManager` associated with the POA. This method may only be invoked after a POA has been created. Attempting to set the `ServantManager` after one has already been set raises a `BAD_INV_ORDER` exception.

This method requires that the `USE_SERVANT_MANAGER` policy be present with the POA; otherwise, a `WrongPolicy` exception is raised.

Parameter	Description
<code>_imagr</code>	The <code>ServantManager</code> to be used as the default used with the POA.

```
PortableServer::AdapterActivator_ptr the_activator();
```

This method returns the `AdapterActivator` associated with the POA. When a POA is created, it does not have an `AdapterActivator` (i.e., the attribute is null). It is system dependent whether a root POA has an activator and the application can assign one as it wishes.

```
void the_activator(PortableServer::AdapterActivator_ptr _val);
```

This method sets the `AdapterActivator` object associated with the POA to the one specified.

Parameter	Description
<code>_val</code>	The <code>ActivatorAdapter</code> to be associated with the POA.

```
char* the_name();
```

This method returns the read-only attribute which identifies the POA relative to its parent. This parent is assigned at POA creation. The name of the root POA is system dependent and should not be relied upon by the application.

```
PortableServer::POA_ptr the_parent();
```

This method returns a pointer to the POA's parent POA. The parent of the root POA is null.

```
PortableServer::POAManager_ptr the_POAManager();
```

This method returns the read-only attribute which is a pointer to the `POAManager` associated with the POA.

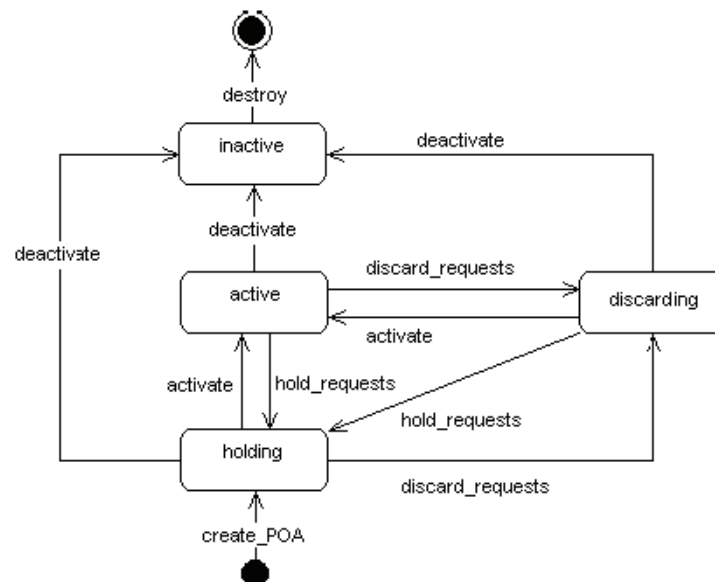
PortableServer::POAManager

Each POA has an associated POA manager which in turn may be associated with one or more POA objects. A POA manager encapsulates the processing state of the POAs with which it is associated.

There are four possible states which a POA manager can be in:

- active
- inactive
- holding
- discarding

A POA manager is created in the holding state. The figure below illustrates the states which a POA manager transitions to based on the method called.



Include file

You should include the file `poa_c.hh` when using this class.

PortableServer::POAManager methods

```
void activate();
```

This method changes the state of the POA manager to active, which enables the associated POAs to process requests. If invoked while the POA manager is in the inactive state, the `AdapterInactive` exception is raised.

```
void deactivate(CORBA::Boolean _etherealize_objects, CORBA::Boolean
_wait_for_completion);
```

This method changes the state of the POA manager to inactive, which causes the associated POAs to reject requests that have not begun execution, as well as any new requests. If invoked while the POA manager is in the inactive state, the `AdapterInactive` exception is raised.

After the state changes, if the `etherealize_objects` parameter is

- `TRUE`—the POA manager causes all associated POAs that have the `RETAIN` and `USE_SERVANT_MANAGER` policies to perform the `etherealize` operation on the associated servant manager for all active objects.
- `FALSE`—the `etherealize` operation is not called. The purpose is to provide developers with a means to shut down POAs in a crisis (for example, unrecoverable error) situation.

If the `wait_for_completion` parameter is `FALSE`, this operation returns immediately after changing the state. If the parameter is `TRUE` and the current thread is not in an invocation context dispatched by some POA belonging to the same VisiBroker ORB as this POA, this operation does not return until there are no actively executing requests in any of the POAs associated with this POA manager (that is, all requests that were started prior to the state change have completed) and, in the case of a `TRUE` `etherealize_objects` parameter, all invocations of `etherealize` have completed for POAs having the `RETAIN` and `USE_SERVANT_MANAGER` policies. If the parameter is `TRUE` and the current thread is in an invocation context dispatched by some POA belonging to the same VisiBroker ORB as this POA, the `BAD_INV_ORDER` exception is raised and the state is not changed.

```
void discard_requests(CORBA::Boolean _wait_for_completion);
```

This method changes the state of the POA manager to discarding, which causes the associated POAs to discard incoming requests. In addition, any requests that have been queued but have not started executing are discarded. When a request is discarded, a `TRANSIENT` system exception is returned to the client. If invoked while the POA manager is in the inactive state, the `AdapterInactive` exception is raised.

If the `wait_for_completion` parameter is `FALSE`, this operation returns immediately after changing the state. If the parameter is `TRUE` and the current thread is not in an invocation context dispatched by some POA belonging to the same VisiBroker ORB as this POA, this operation does not return until either there are no actively executing requests in any of the POAs associated with this POA manager (that is, all requests that were started prior to the state change have completed) or the state of the POA manager is changed to a state other than discarding. If the parameter is `TRUE` and the current thread is in an invocation context dispatched by some POA belonging to the same VisiBroker ORB as this POA, the `BAD_INV_ORDER` exception is raised and the state is not changed.

```
void hold_requests(CORBA::Boolean _wait_for_completion);
```

This method changes the state of the POA manager to holding, which causes the associated POAs to queue incoming requests. Any requests that have been queued but are not executing will continue to be queued while in the holding state. If invoked while the POA manager is in the inactive state, the `AdapterInactive` exception is raised.

If the `wait_for_completion` parameter is `FALSE`, this operation returns immediately after changing the state. If the parameter is `TRUE` and the current thread is not in an invocation context dispatched by some POA belonging to the same VisiBroker ORB as this POA, this operation does not return until there are no actively executing requests in any of the POAs associated with this POA manager (that is, all requests that were started prior to the state change have completed) and, in the case of a `TRUE` `etherealize_objects` parameter, all invocations of `etherealize` have completed for POAs having the `RETAIN` and `USE_SERVANT_MANAGER` policies. If the parameter is `TRUE` and the current thread is in an invocation context dispatched by some POA belonging to the same VisiBroker ORB as this POA the `BAD_INV_ORDER` exception is raised and the state is not changed.

Principal

```
typedef OctetSequence Principal
```

The `Principal` is used to represent the client application on whose behalf a request is being made. An object implementation can accept or reject a bind request, based on the contents of the client's `Principal`.

Note This feature has been deprecated as of VisiBroker 4.0.

Include file

You should include the file `corba.h` when using this typedef.

Principal methods

The `BOA` class provides the `get_principal` method, which returns a pointer to the `Principal` associated with an object. The `Object` class also provides methods for getting and setting the `Principal`.

PortableServer::RefCountServantBase

```
class RefCountServantBase : public ServantBase
```

This class can be used as a standard servant reference counting mix-in class, rather than the `PortableServer::ServantBase` class which is to be used with inheritance class. For more information, see [“PortableServer::ServantBase” on page 43](#).

Include file

You should include the file `poa_c.h` when using this class.

PortableServer::RefCountServantBase methods

```
void _add_ref();
```

This method increments the reference count by one. You can use this method from the base class to provide true reference counting.

```
void _remove_ref();
```

This method decrements the reference count by one. You can override this method from the base class to provide true reference counting.

PortableServer::ServantActivator

```
class PortableServer::ServantActivator : public PortableServer::ServantManager
```

If the POA has the `RETAIN` policy present, then it uses servant managers that are `PortableServer::ServantActivator` objects.

Include file

You should include the file `poa_c.hh` when using this class.

PortableServer::ServantActivator methods

```
void etherealize(PortableServer::ObjectId& oid, PortableServer::POA_ptr adapter,
PortableServer::Servant serv, CORBA::Boolean cleanup_in_progress, CORBA::Boolean
remaining_activations);
```

This method is called by the specified `adapter` whenever a servant for an object (the specified `oid`) is deactivated, assuming that the `RETAIN` and `USE_SERVANT_MANAGER` policies are present.

Parameter	Description
<code>oid</code>	The object id of the object whose servant is to be deactivated.
<code>adapter</code>	The POA in whose scope the object was active.
<code>serv</code>	The servant which is to be deactivated.
<code>cleanup_in_progress</code>	If set to <code>TRUE</code> , the reason for the invocation of the method is either that the <code>deactivate</code> or <code>destroy</code> method was called with the <code>etherealize_objects</code> parameter set to <code>TRUE</code> ; otherwise, the method was called for other reasons.
<code>remaining_activations</code>	If the specified <code>serv</code> is associated with other objects in the specified <code>adapter</code> it is set to <code>TRUE</code> ; otherwise it is <code>FALSE</code> .

```
PortableServer::Servant incarnate(const PortableServer::ObjectId& oid,
PortableServer::POA_ptr adapter);
```

This method is called by the POA whenever the POA receives a request for an inactive object (the specified `oid`) assuming that the `RETAIN` and `USE_SERVANT_MANAGER` policies are present.

The user supplies a servant manager implementation which is responsible for locating and creating an appropriate servant that corresponds to the specified `oid` value. The method returns a servant, which is also entered into the Active Object map. Any further requests for the active object are passed directly to the servant associated with it without invoking the servant manager.

If this method returns a servant that is already active for a different object id and if the POA also has the `UNIQUE_ID` policy present, then it raises the `OBJ_ADAPTER` exception.

Parameter	Description
<code>oid</code>	The object id of the object whose servant is to be activated.
<code>adapter</code>	The POA in whose scope the object is to be activated.

PortableServer::ServantBase

```
class PortableServer::ServantBase
```

The `Portable::ServantBase` class is the base class for your server application.

Include file

You should include the file `poa_c.hh` when using this class.

PortableServer::ServantBase methods

```
void _add_ref();
```

This method adds a reference count for this servant. It should be overridden to provide reference counting functionality for classes derived from this class as the default implementation does nothing.

```
PortableServer::POA_ptr _default_POA();
```

This method returns an Object reference to the root POA of the default VisiBroker ORB in the current process, (i.e., the same return value as an invocation of `ORB::resolve_initial_references("RootPOA")` on the default VisiBroker ORB. Classes derived from the `PortableServer::ServantBase` class may override this method to return the POA of their choice, if desired.

```
CORBA::InterfaceDef_ptr _get_interface();
```

This method returns a pointer to this object's interface definition. See the [Interface repository interfaces and classes, "InterfaceDef methods" on page 97](#), for more information.

```
CORBA::Boolean _is_a(const char *rep_id);
```

This method returns `TRUE` if this servant implements the interface associated with the repository id. Otherwise, it returns `FALSE`.

Parameter	Description
<code>rep_id</code>	The repository identifier against which to check.

```
void _remove_ref();
```

This method removes a reference count for this servant. It should be overridden to provide reference counting functionality for classes derived from this class as the default implementation does nothing.

PortableServer::ServantLocator

```
class PortableServer::ServantLocator : public PortableServer::ServantManager
```

When the POA has the `NON_RETAIN` policy present, it uses servant managers which are `PortableServer::ServantLocator` objects. The servant returned by the servant manager will be used only for a single request.

Because the POA knows that the servant returned by the servant manager will be used only for a single request, it can supply extra information for the servant manager's methods and the servant manager's pair of methods may do something different than a `PortableServer::ServantLocator` servant manager.

Include file

You should include the file `poa_c.hh` when using this class.

PortableServer::ServantLocator methods

```
PortableServer::Servant preinvoke(const PortableServer::ObjectId& oid,
PortableServer::POA_ptr adapter, const char* operation, Cookie& the_cookie);
```

This method is called by the POA whenever the POA receives a request for an object that is not currently active, assuming that the `NON_RETAIN` and `USE_SERVANT_MANAGER` policies are present.

The user-supplied implementation of the servant manager is responsible for locating or creating an appropriate servant that corresponds to the specified `oid` value if possible.

Parameter	Description
<code>oid</code>	The <code>ObjectId</code> value that is associated with the incoming request.
<code>adapter</code>	The POA in which the object is to be activated.
<code>operation</code>	The name of the operation which will be called by the POA when the servant is returned.
<code>the_cookie</code>	An opaque value which can be set by the servant manager to be used later in the <code>postinvoke</code> method.

```
void postinvoke(const PortableServer::ObjectId& oid, PortableServer::POA_ptr
adapter, const char* operation, Cookie the_cookie, PortableServer::Servant
the_servant)
```

If the POA has the `NON_RETAIN` and `USE_SERVANT_MANAGER` policies present, this method is called whenever a servant completes a request. This method is considered to be part of the request on an object, (i.e., if the method finishes normally, but `postinvoke` raises a system exception, then the method's normal return is overridden; and the request completes with the exception).

Destroying a servant that is known to a POA can lead to undefined results.

Parameter	Description
<code>oid</code>	The <code>ObjectId</code> value that is associated with the incoming request.
<code>adapter</code>	The POA in which the object is to be activated.
<code>operation</code>	The name of the operation which will be called by the POA when the servant is returned.

Parameter	Description
<code>the_cookie</code>	An opaque value which can be set by the servant manager in the <code>preinvoke</code> method for use in this method.
<code>the_servant</code>	The servant associated with the object.

PortableServer::ServantManager

```
class PortableServer::ServantManager
```

Servant managers are associated with Portable Object Adapters (POAs). A servant manager allows a POA to activate objects on demand when the POA receives a request targeted for an inactive object.

The `PortableServer::ServantManager` class has no methods; rather it is the base class for two other classes: the `PortableServer::ServantActivator` and the `PortableServer::ServantLocator` classes. For more details, see [“PortableServer::ServantActivator” on page 42](#) and [“PortableServer::ServantLocator” on page 44](#). The use of these two classes depends on the POA's policies: `RETAIN` for the `PortableServer::ServantActivator` and `NON_RETAIN` for the `PortableServer::ServantLocator`.

Include file

You should include the file `poa_c.h` when using this class.

SystemException

```
class CORBA::SystemException : public CORBA::Exception
```

The `SystemException` class is used to report standard system errors encountered by the VisiBroker ORB or by the object implementation. This class is derived from the `Exception` class, described in [“Exception” on page 19](#), which provides methods for printing the name and details of the exception to an output stream.

`SystemException` objects include a completion status which indicates if the operation that caused the exception was completed. `SystemException` objects also have a minor code that can be set and retrieved.

Include file

The `corba.h` file should be included when you use this class.

SystemException methods

```
CORBA::SystemException(CORBA::ULong minor = 0, CORBA::CompletionStatus status = CORBA::COMPLETED_NO);
```

This method creates a `SystemException` object with the specified properties.

Parameter	Description
<code>minor</code>	The minor code.
<code>status</code>	The completion status, one of <code>CORBA::COMPLETED_YES</code> , <code>CORBA::COMPLETED_NO</code> , or <code>CORBA::COMPLETED_MAYBE</code> .

```
CORBA::CompletionStatus completed() const;
```

This method returns `TRUE` if this object's completion status is set to `COMPLETED_YES`.

```
void completed(CORBA::CompletionStatus status);
```

This method sets the completion status for this object.

Parameter	Description
<code>status</code>	The completion status, one of <code>COMPLETED_YES</code> , <code>COMPLETED_NO</code> , or <code>COMPLETED_MAYBE</code> .

```
CORBA::ULong minor() const;
```

This method returns this object's minor code.

```
void minor(CORBA::ULong val);
```

This method sets the minor code for this object.

Parameter	Description
<code>val</code>	The minor code.

```
static CORBA::SystemException *_downcast(CORBA::Exception *exc);
```

This method attempts to downcast the specified `Exception` pointer to a `SystemException` pointer. If the supplied pointer points to a `SystemException` object or an object derived from `SystemException`, a pointer to the object is returned. If the supplied pointer does not point to a `SystemException` object, a `NULL` pointer is returned.

Parameter	Description
<code>exc</code>	An <code>Exception</code> pointer to be down casted.

Note The reference count for the `Exception` object is not incremented by this method.

Exception name	Description
<code>BAD_INV_ORDER</code>	Routine invocations out of order.
<code>BAD_OPERATION</code>	Invalid operation.
<code>BAD_CONTEXT</code>	Error processing context object.
<code>BAD_PARAM</code>	An invalid parameter was passed.
<code>BAD_TYPECODE</code>	Invalid typecode.
<code>COMM_FAILURE</code>	Communication failure.
<code>DATA_CONVERSION</code>	Data conversion error.
<code>FREE_MEM</code>	Unable to free memory.
<code>IMP_LIMIT</code>	Implementation limit violated.
<code>INITIALIZE</code>	ORB initialization failure.
<code>INTERNAL</code>	ORB internal error.
<code>INTF_REPOS</code>	Error accessing interface repository.
<code>INV_FLAG</code>	Invalid flag was specified.
<code>INV_INDENT</code>	Invalid identifier syntax.
<code>INV_OBJREF</code>	Invalid object reference specified.
<code>MARSHAL</code>	Error marshalling parameter or result.
<code>NO_IMPLEMENT</code>	Operation implementation not available.
<code>NO_MEMORY</code>	Dynamic memory allocation failure.
<code>NO_PERMISSION</code>	No permission for attempted operation.

Exception name	Description
NO_RESOURCES	Insufficient resources to process request.
NO_RESPONSE	Response to request not yet available.
OBJ_ADAPTOR	Failure detected by object adaptor.
OBJECT_NOT_EXIST	Object is not available.
PERSIST_STORE	Persistent storage failure.
TRANSIENT	Transient failure.
UNKNOWN	Unknown exception.

Dynamic interfaces and classes

The `CORBA::Any` class is used to represent an IDL type so that its value may be passed in a type-safe manner. Objects of this class have a pointer to a `TypeCode` that defines the object's type and a pointer to the value associated with the object. Methods are provided to construct, copy, and destroy an object as well as to initialize and query the object's type and value. In addition, streaming operators are provided to read and write the object to a stream.

The code sample below provides an example of how to create and use an `Any`.

```
// create an any object
CORBA::Any anObject;
// use the typeid operator to specify that
// 'anObject' object can store long
anObject <<= CORBA::_tc_long;
```

Include file

Include the **CORBA.h** file when you use this structure.

Any methods

```
CORBA::Any();
```

This is the default constructor. It creates an empty `Any` object.

```
CORBA::Any(const CORBA::Any& val);
```

This is a copy constructor; it creates an `Any` object that is a copy of the specified target.

Parameter	Description
<code>val</code>	The object to be copied.

```
CORBA::Any(CORBA::TypeCode_ptr tc, void *value, CORBA::Boolean release = 0);
```

This constructor creates an `Any` object initialized with the specified value and `TypeCode`.

Parameter	Description
<code>tc</code>	The <code>TypeCode</code> of the value contained by this <code>Any</code> .
<code>value</code>	The value contained by this <code>Any</code> .
<code>release</code>	If set to <code>TRUE</code> , the memory associated with this <code>Any</code> object's value is released when this <code>Any</code> object is destroyed.

```
static CORBA::Any_ptr _duplicate(CORBA::Any_ptr ptr);
```

This static method increments the reference count for the specified object and then returns a pointer to it.

Parameter	Description
<code>ptr</code>	The <code>Any</code> to be duplicated.

```
static CORBA::Any_ptr _nil();
```

This static method returns a `NULL` pointer that can be used for initialization purposes.

```
static void _release(CORBA::Any_ptr *ptr);
```

This static method decrements the reference count for the specified object. When the count reaches zero, all memory managed by the object is released and the object is deleted.

Parameter	Description
<code>ptr</code>	The <code>Any</code> to be released.

Insertion operators

```
void operator<<=(CORBA::Short);
void operator<<=(CORBA::UShort);
void operator<<=(CORBA::Long);
void operator<<=(CORBA::ULong);
void operator<<=(CORBA::Float);
void operator<<=(CORBA::Double);
void operator<<=(const CORBA::Any&);
void operator<<=(const char *);
void operator<<=(CORBA::LongLong);
void operator<<=(CORBA::ULongLong);
void operator<<=(CORBA::LongDouble);
```

These operators initialize this object with the specified value, automatically setting the appropriate `TypeCode` for the value. If this `Any` object was constructed with the `release` flag set to `TRUE`, the value previously stored in this `Any` object is released before the new value is assigned.

```
void operator<<=(CORBA::TypeCode_ptr tc);
```

This method initializes this object with the specified `TypeCode` of the value.

Parameter	Description
<code>tc</code>	The <code>TypeCode</code> to set for this <code>Any</code> .

Extraction operators

```

CORBA::Boolean operator>>=(CORBA::Short&) const;
CORBA::Boolean operator>>=(CORBA::UShort&) const;
CORBA::Boolean operator>>=(CORBA::Long&) const;
CORBA::Boolean operator>>=(CORBA::ULong&) const;
CORBA::Boolean operator>>=(CORBA::Float&) const;
CORBA::Boolean operator>>=(CORBA::Double&) const;
CORBA::Boolean operator>>=(CORBA::Any&) const;
CORBA::Boolean operator>>=(char *&) const;
CORBA::Boolean operator>>=(CORBA::LongLong&) const;
CORBA::Boolean operator>>=(CORBA::ULongLong&) const;
CORBA::Boolean operator>>=(CORBA::LongDouble&) const;

```

These operators store the value from this object into the specified target. If the `TypeCode` of the target does not match the `TypeCode` of the stored value, `FALSE` is returned and no value is extracted. Otherwise, the stored value is assigned to the target and `TRUE` is returned.

```
CORBA::Boolean operator>>=(CORBA::TypeCode_ptr& tc) const;
```

This method extracts the `TypeCode` of the value stored in this object.

Parameter	Description
<code>tc</code>	The object where the <code>TypeCode</code> for this <code>Any</code> is to be stored

ContextList

```
class CORBA::ContextList
```

This class contains a list of contexts that may be associated with an operation request. See [“Request” on page 69](#) for more information.

ContextList methods

```
CORBA::ContextList();
```

This method constructs an empty `Context` list.

```
~CORBA::ContextList();
```

This method is the default destructor.

```
void add(const char *ctx);
```

This method adds the specified context to this object's list.

Parameter	Description
<code>ctx</code>	The context to be added to the list.

```
void add_consume(char *ctx);
```

This method adds the specified context code to this object's list. This `ContextList` becomes the owner of the context specified by the argument. You should not attempt to access or free this `Context` after you invoke this method.

Parameter	Description
<code>ctx</code>	The context to be added to the list.

```
CORBA::ULong count() const;
```

This method returns the number of items currently stored in the list.

```
const char *item(CORBA::Long index);
```

This method returns a pointer to the context that is stored in the list at the specified index. If the index is invalid, a `NULL` pointer is returned. You should not attempt to free the returned context. To remove a context, use the `remove` method instead.

Parameter	Description
<code>index</code>	The zero-based index of the context to be returned.

```
void remove(CORBA::long index);
```

This method removes the context with the specified index from the list. If the index is invalid, no removal will occur.

Parameter	Description
<code>index</code>	The zero-based index of the context to be removed.

```
static CORBA::ContextList_ptr _duplicate(CORBA::ContextList_ptr ptr);
```

This static method increments the reference count for the object and then returns a pointer to it.

Parameter	Description
<code>ptr</code>	The object to be duplicated.

```
static CORBA::ContextList_ptr _nil();
```

This static method returns a `NULL` pointer that can be used for initialization purposes.

```
static void _release(CORBA::ContextList *ptr);
```

This static method decrements the reference count for this object. When the count reaches zero, all memory managed by the object is released and the object is deleted.

Parameter	Description
<code>ptr</code>	The object to be released.

DynamicImplementation

```
class PortableServer::DynamicImplementation : public PortableServer::ServantBase
```

This base class is used derive object implementations that use the Dynamic Skeleton Interface instead of a skeleton class generated by the IDL compiler. You must provide implementations of the `invoke` and `_primary-interface()` methods when deriving from this class.

DynamicImplementation methods

```
virtual void invoke(CORBA::ServerRequest_ptr request) = 0;
```

This method is invoked by the POA whenever client operation requests are received for your object implementation. You must provide an implementation of this method which validates the `ServerRequest` object's contents, performs the necessary processing to fulfill the request, and returns the results to the client. For more information on the `ServerRequest` class, see [“ServerRequest” on page 72](#).

Parameter	Description
<code>request</code>	The <code>ServerRequest</code> object that represents the client's operation request.

```
virtual CORBA::RepositoryId _primary_interface(const PortableServer::ObjectId& oid
PortableServer::POA_ptr poa) const;
```

This method will be invoked as a callback by the POA. The servants that inherit from the `DynamicImplementation` class must implement it. This method should be called directly or unpredictable behavior will result. Invoking this method under other circumstances may lead to unpredictable results. The `_primary_interface` method receives an `ObjectId` value and a `POA_ptr` as input parameters and returns a valid `RepositoryId` representing the most-derived interface for that `oid`.

DynAny

```
class DynamicAny::DynAny : public CORBA::Pseudo Object
```

A `DynAny` object is used by a client application or server to create and interpret data types at runtime which were not defined at compile time. A `DynAny` may contain a basic type (such as a `boolean`, `int`, or `float`) or a complex type (such as a `struct` or `union`). The type contained by a `DynAny` is defined when it is created and may not be changed during the lifetime of the object.

A `DynAny` object may represent a data type as one or more components, each with its own value. The `next`, `seek`, `rewind`, and `current_component` methods are provided to help you navigate through the components.

A `DynAnyFactory` is created by calling `ORB::resolve_initial_references("DynAnyFactory")`. The factory is then used to create basic or complex types. The `DynAnyFactory` belongs to the `DynamicAny` module.

`DynAny` objects for basic types are created using the `DynAnyFactory::create_dyn_any_from_type_code` method. A `DynAny` object may also be created and initialized from an `Any` object using the `DynAnyFactory::create_dyn_any` method.

The following interfaces are derived from `DynAny` and provide support for constructed types that are managed dynamically.

Constructed type	Interface
Array	<code>DynArray</code> in “ DynArray ” on page 57.
Enumeration	<code>DynEnum</code> in “ DynEnum ” on page 58.
Sequence	<code>DynSequence</code> in “ DynSequence ” on page 59.
Structure	<code>DynStruct</code> in “ DynStruct ” on page 60.
Union	<code>DynUnion</code> in “ DynUnion ” on page 61.

Include file

The `dynany.h` file should be included when you use this class.

Important usage restrictions

`DynAny` objects cannot be used as parameters on operation requests or DII requests, nor can they be externalized using the `ORB::object_to_string` method. However, you may use the `DynAny::to_any` method to convert a `DynAny` object into an `Any`, which can be used as a parameter.

DynAny methods

```
void assign(DynamicAny::DynAny_ptr dyn_any);
```

Initializes the value in this `DynAny` object from the specified `DynAny`.

A type mismatch exception is raised if the type contained in the `Any` does not match the type contained by this object.

```
DynamicAny::DynAny_ptr copy();
```

Returns a copy of this object.

```
virtual CORBA::ULong component_count();
```

Returns the number of components for the complex type stored inside the `DynAny` as an unsigned long.

```
virtual DynamicAny::DynAny_ptr current_component();
```

Returns the current component in this object.

```
virtual void destroy();
```

Destroys this object.

```
virtual CORBA::Boolean equal(const DynamicAny::DynAny_ptr value);
```

Compares two `DynAny` values for equality. Returns `TRUE` if they are equal, `FALSE` otherwise.

```
virtual void from_any(CORBA::Any& value);
```

Initializes the current component of this object from the specified Any object.

A type mismatch exception is raised if the `TypeCode` of value contained in the Any does not match the `TypeCode` that was defined for this object when it was created.

If the value parameter passed is not legal, the operation raises an `InvalidValue` exception.

Parameter	Description
value	An Any object containing the value to set for this object.

```
virtual boolean next();
```

Advances to the next component, if one exists, and returns `TRUE`. If there are no more components, this method returns `FALSE`.

```
virtual void rewind();
```

Sets the current component of this object to be the first component defined in this `DynAny`.

If this object contains only one component, invoking this method has no effect.

```
virtual CORBA::Boolean seek(CORBA::Long index);
```

Makes the component with the specified index the current component. If there is no component at the specified index, this method returns `FALSE`, otherwise it returns `TRUE`.

Parameter	Description
index	The zero-based index of the desired component.

```
virtual CORBA::Any* to_any( );
```

Converts the `DynAny` object into an Any object and returns a pointer to the Any object.

```
CORBA::TypeCode_ptr type();
```

Returns the `TypeCode` of the value stored in the `DynAny`.

Extraction methods

The `DynAny` extraction methods return the type contained in this `DynAny` object's current component. The list below shows the name of each of the extraction methods.

A `TypeMismatch` exception is raised if the value contained in this `DynAny` does not match the expected return type for the extraction method used.

Extraction methods offered by the `DynAny` class are:

```
virtual CORBA::Any* get_any();
virtual CORBA::Boolean get_boolean();
virtual CORBA::Char get_char();
virtual CORBA::Double get_double();
virtual DynamicAny::DynAny* get_dyn_any();
virtual CORBA::Float get_float();
virtual CORBA::Long get_long();
virtual CORBA::Long get_longlong();
virtual CORBA::Octet get_octet();
virtual CORBA::Object_ptr get_reference();
```

```

virtual CORBA::Short get_short();
virtual char* get_string();
virtual CORBA::TypeCode_ptr get_typecode();
virtual CORBA::ULong get_ulong();
virtual CORBA::ULongLong get_ulonglong();
virtual CORBA::UShort get_ushort();
virtual CORBA::ValueBase* get_val();
virtual CORBA::WChar get_wchar();
virtual CORBA::WChar* get_wstring();

```

Solaris:

```

virtual CORBA::LongDouble get_longdouble();

```

Insertion methods

An insertion method copies a value of a particular type to this `DynAny` object's current component. Following is the list of methods provided for inserting various types.

These methods raise an `InvalidValue` exception if the inserted object's type does not match the `DynAny` object's type.

Insertion methods offered by the `DynAny` class are:

```

virtual void insert_any(const CORBA::Any& value);
virtual void insert_boolean(CORBA::Boolean value);
virtual void insert_char(CORBA::char value);
virtual void insert_double(CORBA::Double value);
virtual void insert_dyn_any (DynamicAny::DynAny_ph_value);
virtual void insert_float(CORBA::Float value);
virtual void insert_long(CORBA::Long value);
virtual void insert_longlong(CORBA::LongLong value);
virtual void insert_octet(CORBA::Octet value);
virtual void insert_reference(CORBA::Object_ptr value);
virtual void insert_short(CORBA::Short value);
virtual void insert_string(const char* value);
virtual void insert_typecode(CORBA::TypeCode_ptr value);
virtual void insert_ulong(CORBA::ULong value);
virtual void insert_ulonglong(CORBA::ULongLong value);
virtual void insert_ushort(CORBA::UShort value);
virtual void insert_val(count CORBA::ValueBase& value); virtual void
    insert_wchar(CORBA::WChar value);
virtual void insert_wstring(const CORBA::WChar* value);

```

Solaris:

```

virtual void insert_longdouble(CORBA::LongDouble value); Solaris only

```


DynAnyFactory

```
class DynamicAny::DynAnyFactory : public CORBA::PseudoObject
```

A `DynAnyFactory` object is used to create a new `DynAny` object. To obtain a reference to the `DynAnyFactory` object, call `ORB::resolve_initial_references("DynAnyFactory")`.

DynAnyFactory methods

```
DynAny_ptr create_dyn_any (const CORBA::Any& value);
```

Creates a `DynAny` object of the specified value

Parameter	Description
value	A new <code>DynAny</code> object of a specified value.

```
DynAny_ptr create_dyn_any_from_type_code (CORBA::TypeCode_ptr type);
```

Creates a `DynAny` object of the specified type.

Parameter	Description
type	The type of the new <code>DynAny</code> object.

DynArray

```
class DynamicAny::DynArray : public VISDynComplex
```

Objects of this class are used by a client application or server to create and interpret array data types at runtime which were not defined at compile time. A `DynArray` may consist of a sequence of basic types (such as a `boolean`, `int`, or `float`) or constructed types (such as `struct` or `union`). The type contained by a `DynArray` is defined when it is created and may not be changed during the lifetime of the object.

The `next`, `rewind`, `seek`, and `current_component` methods, inherited from `DynAny`, may be used to navigate through the components.

The `VISDynComplex` class is a helper class that allows the `VisiBroker` ORB to manage complex `DynAny` types.

Important usage restrictions

`DynArray` objects cannot be used as parameters on operation requests or DII requests, nor can they be externalized using the `ORB::object_to_string` method. However, you may use the `DynAny::to_any` method to convert a `DynArray` object to a sequence of `Any` objects, which can be used as a parameter.

DynArray methods

```
virtual void destroy();
```

Destroys this object.

```
CORBA::AnySeq* get_elements();
```

Returns a sequence of `Any` objects containing the values stored in this object.

```
void set_elements(CORBA::AnySeq& _value);
```

Assigns the elements in the `DynArray` to those in the sequence specified by the `value` parameter.

```
DynamicAny::DynAnySeq* get_elements_as_dyn_any();
```

Returns the elements contained in the `DynAny` as a `DynAny` sequence.

```
void set_elements_as_dyn_any (const DynamicAny::DynAnySeq& value);
```

Sets the elements contained in the object from the specified `DynAny` sequence.

An `InvalidValue` exception is raised if the number of elements in `value` is not equal to the number of elements in this `DynArray`. A type mismatch exception is raised if the type of the `Any` values do not match the `TypeCode` of the `DynAny`.

Parameter	Description
<code>_value</code>	An array of <code>Any</code> objects whose values will be set in this <code>DynArray</code> .

DynEnum

```
class DynamicAny::DynEnum : public DynamicAny::DynAny
```

Objects of this class are used by a client application or server to create and interpret enumeration values at runtime which were not defined at compile time.

Since objects of this type contains a single component, the `DynAny::rewind` and `DynAny::next` methods of a `DynEnum` object always return `FALSE`.

Important usage restrictions

`DynEnum` objects cannot be used as parameters on operation requests or DII requests, nor can they be externalized using the `ORB::object_to_string` method. However, you may use the `to_any` method to convert a `DynEnum` object to an `Any`, which can be used as a parameter.

DynEnum methods

```
void from_any(const CORBA::Any& value);
```

Initializes the value of this object using the specified `Any` object.

An `Invalid` exception is raised if the `TypeCode` of `value` contained in the `Any` does not match the `TypeCode` defined for this object when it was create.

Parameter	Description
<code>value</code>	An <code>Any</code> object.

```
CORBA::Any* to_any();
```

Returns an `Any` object containing the value of the current component.

```
>char* get_as_string();
```

Returns the `DynEnum` object's value as a string.

```
void set_as_string(const char* value_as_string);
```

Sets the value of this `DynEnum` to the specified string.

Parameter	Description
<code>value_as_string</code>	A string that will be used to set the value in this <code>DynEnum</code> .

```
CORBA::ULong get_as_ulong();
```

Returns an unsigned long containing the `DynEnum` object's value.

```
void set_as_ulong(CORBA::ULong value_as_ulong)
```

Sets the value of this `DynEnum` to the specified `CORBA::ULong`.

Parameter	Description
<code>value_as_ulong</code>	An integer that will be used to set the value in this <code>DynEnum</code> .

DynSequence

```
class DynamicAny::DynSequence : public DynamicAny::DynArray
```

Objects of this class are used by a client application or server to create and interpret sequence data types at runtime which were not defined at compile time. A `DynSequence` may contain a sequence of basic types (such as a `boolean`, `int`, or `float`) or constructed types (such as a `struct` or `union`). The type contained by a `DynSequence` is defined when it is created and may not be changed during the lifetime of the object.

The `next`, `rewind`, `seek`, and `current_component` methods may be used to navigate through the components.

Important usage restrictions

`DynSequence` objects cannot be used as parameters on operation requests or DII requests nor can they be externalized using the `ORB::object_to_string` method. However, you may use the `to_any` method to convert a `DynSequence` object to a sequence of `Any` objects. You can use the sequence of `Any` objects as a parameter.

DynSequence methods

```
CORBA::ULong get_length();
```

Returns the number of elements contained in this `DynSequence`.

```
void set_length(CORBA::ULong length);
```

Sets the number of elements contained in this `DynSequence`.

If you specify a length that is less than the current number of elements, the sequence is truncated.

Parameter	Description
<code>length</code>	The number of components to be contained in this <code>DynSequence</code> .

```
CORBA::AnySeq * get_elements();
```

Returns a sequence of `Any` objects containing the value stored in this object.

```
void set_elements (const AnySeq& _value)
```

Sets the elements within this object with specified sequence of `Any` objects.

```
set _elements_as_dyn_any();
```

See [“DynArray” on page 57](#) for more details.

```
get_elements_as_dyn_any();
```

See [“DynArray” on page 57](#) for more details.

DynStruct

```
class DynamicAny::DynStruct :public VISDynComplex
```

Objects of this class are used by a client application or server to create and interpret structures at runtime which were not defined at compile time.

The `next`, `rewind`, `seek`, and `current_component` methods may be used to navigate through the structure members.

You create an `DynStruct` object by invoking the `DynAnyFactory::create_dyn_any_from_typecode` method.

Important usage restrictions

`DynStruct` objects cannot be used as parameters on operation requests or DII requests, nor can they be externalized using the `ORB::object_to_string` method. However, you may use the `to_any` method to convert a `DynStruct` object to an `Any` object, which can be used as a parameter.

DynStruct methods

```
void destroy();
```

Destroys this object.

```
CORBA::FieldName current_member_name();
```

Returns the member name of the current component.

```
CORBA::TCKind current_member_kind();
```

Returns the `TypeCode` associated with the current component.

```
DynamicAny::NameValuePairSeq get_members();
```

Returns the members of the structure as a sequence of `NameValuePair` objects.

```
void set_members(const DynamicAny::NameValuePairSeq& value);
```

Sets the structure members from the array of `NameValuePair` objects.

```
DynamicAny::Name DynAnyPairSeq get_members_as_dyn_any();
```

Returns the members of the structure as a `NameDynAnyPair` sequence.

```
void set_members_as_dyn_any(const DynamicAny::nameDynAnyPairSeq value);
```

Sets the structure members from `NameDynAnyPair` objects.

An `InvalidValue` exception is raised if the length of the value sequence is not equal to the number of members of `DynStruct`, and a `TypeMismatch` exception is raised when any of the element's typecode does not match that of the structure.

DynUnion

```
class DynamicAny::DynUnion : public VISDynComplex
```

This interface is used by a client application or server to create and interpret unions at runtime which were not defined at compile time. The `DynUnion` contains a sequence of two elements: the union discriminator and the actual member.

The `next`, `rewind`, `seek`, and `current_component` methods may be used to navigate through the components.

You create a `DynUnion` object by invoking the

`DynamicAny::DynAnyFactory::create_dyn_any_from_type_code` method and passing a union type code as an argument.

Important usage restrictions

`DynUnion` objects cannot be used as parameters on operation requests or DII requests nor can they be externalized using the `ORB::object_to_string` method. However, you may use the `DynAny::to_any` method to convert a `DynUnion` object to an `Any` object which can be used as a parameter.

DynUnion methods

```
DynamicAny::DynAny_ptr get_discriminator();
```

Returns a `DynAny` object containing the discriminator for the union.

```
CORBA::TCKind discriminator_kind();
```

Returns the type code of the discriminator for the union.

```
DynamicAny::DynAny_ptr member();
```

Returns a `DynAny` object for the current component which represents a union member.

```
CORBA::TCKind member_kind();
```

Returns the type code for the current component, which represents a member in the union.

```
CORBA::FieldName member_name();
```

Returns the member name of the current component.

```
void set_discriminator (DynamicAny::DynAny_ptr value);
```

Sets the discriminator of this `DynUnion` to the specified value.

```
void set_to_default_member();
```

Sets the discriminator to a value that is consistent with the value of the default case of a union.

```
void set_to_no_active_member();
```

Sets the discriminator to a value that does not correspond to any of the union's case labels.

```
boolean has_no_active_member();
```

Returns `TRUE` if the union has no active member (that is, the union's value consists solely of its discriminator because the discriminator has a value that is not listed as an explicit case label).

Environment

```
class CORBA::Environment
```

The `Environment` class is used for reporting and accessing both system and user exceptions on platforms where C++ language exceptions are not supported. When an interface specifies that user exceptions may be raised by the object's methods, the `Environment` class becomes an explicit parameter of that method. If an interface does not raise any exceptions, the `Environment` class is an implicit parameter and is only used for reporting system exceptions. If an `Environment` object is not passed from the client to a stub, the default of per-object `Environment` is used.

Multithreaded applications have a global `Environment` object for each thread that is created. Applications that are not multithreaded have just one global `Environment` object.

Include file

You should include the `corba.h` file when you use this class.

Environment methods

```
CORBA::Status ORB::create_environment(CORBA::Environment_ptr& ptr);
```

This method can be used to create a new `Environment` object.

Note This method is provided for CORBA compliance. You may find it easier to use the constructor provided for this class or the C++ `new` operator.

Parameter	Description
<code>ptr</code>	The pointer will be set to point to the newly created object.

```
Environment();
```

This method creates an `Environment` object. This is equivalent to calling the `ORB::create_environment` method.

```
static CORBA::Environment& CORBA::current_environment();
```

This static method returns a reference to the global `Environment` object for the application process. In multithreaded applications, it returns the global `Environment` object for this thread.

```
void exception(CORBA::Exception *exp);
```

This method records the `Exception` object passed as an argument. The `Exception` object must be dynamically allocated because the specified object will assume ownership of the `Exception` object and will delete it when the `Environment` itself is deleted. Passing a `NULL` pointer to this method is equivalent to invoking the `clear` method on the `Environment`.

Parameter	Description
<code>exp</code>	A pointer to a dynamically allocated <code>Exception</code> object to be recorded for this <code>Environment</code> .

```
CORBA::Exception *exception() const;
```

This method returns a pointer to the `Exception` currently recorded in this `Environment`. You must not invoke `delete` on the `Exception` pointer returned by this call. If no `Exception` has been recorded, a `NULL` pointer is returned.

```
void clear();
```

This method deletes any `Exception` object that it holds. If this object holds no exception, this method has no effect.

ExceptionList

```
class CORBA::ExceptionList
```

This class contains a list of type codes that represent exceptions that may be raised by an operation request. See [“Request” on page 69](#).

ExceptionList methods

```
CORBA::ExceptionList();
```

This method constructs an empty exception list.

```
CORBA::ExceptionList(CORBA::ExceptionList& list);
```

This is a copy constructor.

Parameter	Description
<code>list</code>	The list to be copied.

```
~CORBA::ExceptionList();
```

This method is the default destructor.

```
void add(CORBA::TypeCode_ptr tc);
```

This method adds the specified exception type code to this object's list.

Parameter	Description
tc	The type code of an exception to be added to the list.

```
void add_consume(CORBA::TypeCode_ptr tc);
```

This method adds the specified exception type code to this object's list. Ownership of the passed argument is assumed by this `ExceptionList`. You should not attempt to access or free the argument after invoking this method.

Parameter	Description
tc	The type code of an exception to be added to the list.

```
CORBA::ULong count() const;
```

This method returns the number of items currently stored in the list.

```
CORBA::TypeCode_ptr item(CORBA::Long index);
```

This method returns a pointer to the `TypeCode` stored in the list at the specified index. If the index is invalid, a `NULL` pointer is returned. You should not attempt to access or free the argument after invoking this method. To remove a `TypeCode` from the list, use the `remove` method.

Parameter	Description
index	The zero-based index of the type code to be returned.

```
void remove(CORBA::long index);
```

This method removes the `TypeCode` with the specified `index` from the list. If the `index` is invalid, no removal occurs.

Parameter	Description
index	The index of the type code to be removed. The index is zero-based.

```
static CORBA::ExceptionList_ptr _duplicate(CORBA::ExceptionList_ptr ptr);
```

This static method increments the reference count for the specified object and then returns a pointer to that object.

Parameter	Description
ptr	The object to be duplicated.

```
static CORBA::ExceptionList_ptr _nil();
```

This static method returns a `NULL` pointer that can be used for initialization purposes.


```
static void _release(CORBA::ExceptionList *ptr);
```

This static method decrements the reference count for the specified object. When the count reaches zero, all memory managed by the object is released and the object is deleted.

Parameter	Description
<code>ptr</code>	The object to be released.

NamedValue

```
class CORBA::NamedValue
```

The `NamedValue` class is used to represent a name-value pair used as a parameter or return value in a Dynamic Invocation Interface request. Objects of this class are grouped into an `NVList`, described in “[NVList](#)” on page 66. The value of the name-value pair is represented by using an `Any` object. The `Request` class is described in “[Request](#)” on page 69.

Include file

You should include the file `corba.h` when using this class.

NamedValue methods

```
CORBA::Flags flags() const;
```

This method returns the flag defining how this name-value pair is to be used. It returns one of the following:

- `ARG_IN` The name-value pair is used as an input parameter.
- `ARG_OUT` The name-value pair is used as an output parameter.
- `ARG_INOUT` The name-value pair is used both as an input and an output parameter.
- `IN_COPY_VALUE` When combined with the `ARG_INOUT` flag, this flag indicates that the ORB copies the output parameter. This allows the ORB to release memory associated with this parameter without impacting the client application's memory.

```
const char *name() const;
```

This method returns the name portion of this object's name-value pair. You should never release the storage pointed to by the return argument.

```
CORBA::Any *value() const;
```

This method returns the value portion of this object's name-value pair. You should never release the storage pointed to by the return argument.

```
static CORBA::NamedValue_ptr _duplicate(CORBA::NamedValue_ptr ptr);
```

This static method increments the reference count for the specified object and then returns a pointer to it.

Parameter	Description
<code>ptr</code>	The object to be duplicated.

```
static CORBA::NamedValue_ptr _nil();
```

This static method returns a NULL pointer that can be used to initialize a `CORBA::NamedValue_ptr`.

```
static void _release(CORBA::NamedValue *ptr);
```

This static method decrements the reference count for the specified object. When the count reaches zero, all memory managed by the object is released and the object is deleted.

Parameter	Description
<code>ptr</code>	The object to be released.

NVList

```
class CORBA::NVList
```

The `NVList` class is used to contain a list of `NamedValue` objects, described in [“NamedValue” on page 65](#). It is used to pass parameters associated with a Dynamic Invocation Interface request. The `Request` class is described in [“Request” on page 69](#).

Several methods are provided for adding items to the list. You should never release the storage pointed to by the return argument. Always use the `remove` method to delete an item from the list.

Include file

You should include the file `corba.h` when using this class.

NVList methods

```
CORBA::NamedValue_ptr add(CORBA::Flags flags);
```

This method adds a `NamedValue` object to this list, initializing only the flags. Neither the name or value of the added object are initialized. A pointer is returned which can be used to initialize the name and value attributes of the `NamedValue`. You should never release the storage associated with the return argument.

Parameter	Description
<code>flags</code>	The flag indicating the intended use of the <code>NamedValue</code> object. It can be one of <code>ARG_IN</code> , <code>ARG_OUT</code> , or <code>ARG_INOUT</code> .

```
CORBA::NamedValue_ptr add_item(const char *name, CORBA::Flags flag);
```

This method adds a `NamedValue` object to this list, initializing the object's `flag` and `name` attributes. A pointer is returned which can be used to initialize the value attribute of the `NamedValue`.

Caution You should never release the storage associated with the return argument.

Parameter	Description
<code>name</code>	The name.
<code>flag</code>	The flag indicating the intended use of the <code>NamedValue</code> object. It can be one of <code>ARG_IN</code> , <code>ARG_OUT</code> , or <code>ARG_INOUT</code> .

```
NamedValue_ptr add_item_consume(char *nm, CORBA::Flags flag);
```

This method is the same as the `add_item` method, except that the `NVList` takes over the management of the storage pointed to by `nm`. You will not be able to access `nm` after this method is called because the list may have copied and released it. When this item is removed, the storage associated with it is automatically freed.

Caution You should never release the memory associated with this method's return value.

Parameter	Description
<code>name</code>	The name.
<code>flag</code>	The flag indicating the intended use of the <code>NamedValue</code> object. It must be one of <code>ARG_IN</code> , <code>ARG_OUT</code> , or <code>ARG_INOUT</code> .

```
CORBA::NamedValue_ptr add_value(const char *name, const CORBA::Any *value,
CORBA::Flags flag);
```

This method adds a `NamedValue` object to this list, initializing the name, value, and flag. A pointer to the `NamedValue` object is returned.

Caution You should never release the storage associated with the return argument.

Parameter	Description
<code>name</code>	The name.
<code>value</code>	The value.
<code>flag</code>	The flag indicating the intended use of the <code>NamedValue</code> object. It can be one of <code>ARG_IN</code> , <code>ARG_OUT</code> , or <code>ARG_INOUT</code> .

```
NamedValue_ptr add_value_consume(char *nm, CORBA::Any *value, CORBA::Flags flag);
```

This method is the same as the `add_value` method, except that the `NVList` takes over the management of the storage pointed to by `nm` and `value`. You will not be able to access `nm` or `value` after this method is called because the list may have copied and released them. When this list element is removed, the storage associated with it is automatically freed.

Parameter	Description
<code>nm</code>	The name.
<code>value</code>	The value.
<code>flag</code>	The flag indicating the intended use of the <code>NamedValue</code> object. It must be one of <code>ARG_IN</code> , <code>ARG_OUT</code> , or <code>ARG_INOUT</code> .

```
CORBA::Long count() const;
```

This method returns the number of `NamedValue` objects in this list.

```
static CORBA::Boolean CORBA::is_nil(NVList_ptr obj);
```

This method returns `TRUE` if the specified `NamedValue` pointer is `NULL`.

Parameter	Description
<code>obj</code>	The pointer to the object to be checked.

```
NamedValue_ptr item(CORBA::Long index);
```

This method returns the `NamedValue` in the list with the specified `index`.

Caution Never release the storage associated with the return argument.

Parameter	Description
<code>index</code>	The zero-based index of the desired <code>NamedValue</code> object.

```
static void CORBA::release(CORBA::NVList_ptr obj);
```

This static method releases the specified object.

Parameter	Description
<code>obj</code>	The object to be released.

```
Status remove(CORBA::Long index);
```

This method deletes the `NamedValue` object located at the specified `index` from this list. Storage associated with items in the list that were added using the `add_item_consume` or `add_value_consume` methods is released before the item is removed.

Parameter	Description
<code>index</code>	The index of the <code>NamedValue</code> object. Note that indexing is zero-based.

```
static CORBA::NVList_ptr _duplicate(CORBA::NVList_ptr ptr);
```

This static method increments the reference count for the specified object and then returns a pointer to that object.

Parameter	Description
<code>ptr</code>	The object to be duplicated.

```
static CORBA::NVList_ptr _nil();
```

This static method returns a `NULL` pointer that can be used to initialize an `NV_List` pointer. For example, you might do something like this: `CORBA::NV_List_ptr p = CORBA::NVList::_nil();`

```
static void _release(CORBA::NVList *ptr);
```

This static method decrements the reference count for the specified object. When the count reaches zero, all memory managed by the object is released and the object is deleted.

Parameter	Description
<code>ptr</code>	The object to be released.

Request

```
class CORBA::Request
```

The `Request` class is used by client applications to invoke an operation on an ORB object using the Dynamic Invocation Interface. A single ORB object is associated with a given `Request` object. The `Request` represents an operation that is to be performed on the ORB object. It includes the arguments to be passed, the `Context`, and an `Environment` object, if any. Methods are provided for invoking the request, receiving the response from the object implementation, and retrieving the result of the operation.

You can create a `Request` object by using the `Object::_create_request`. For more information, go to the *Core interfaces and classes*, [CORBA::Object methods](#) section.

Note that a `Request` object retains ownership of all return parameters, so you should never attempt to free them.

Include file

Include the `corba.h` file when you use this class.

Request methods

```
CORBA::Any& add_in_arg();
```

This method adds an unnamed input argument to this `Request` and returns a reference to the `Any` object so that you can set its name, type, and value.

```
CORBA::Any& add_in_arg(const char *name);
```

This method adds a named input argument to this `Request` and returns a reference to the `Any` object so that you can set its type and value.

Caution You should never release the memory associated with this method's return value.

Parameter	Description
<code>name</code>	The name of the input argument to be added.

```
CORBA::Any& add_inout_arg();
```

This method adds an unnamed `inout` argument to this `Request` and returns a reference to the `Any` object so that you can set its name, type, and value.

```
CORBA::Any& add_inout_arg(const char *name);
```

This method adds a named `inout` argument to this `Request` and returns a reference to the `Any` object so that you can set its type and value.

Parameter	Description
<code>name</code>	The name of the inout argument to be added.

```
CORBA::Any& add_out_arg();
```

This method adds an unnamed output argument to this `Request` and returns a reference to the `Any` object so that you can set its name, type, and value.

```
CORBA::Any& add_out_arg(const char *name);
```

This method adds a named output argument to this `Request` and returns a reference to the `Any` object so that you can set its type and value.

Parameter	Description
<code>name</code>	The name of the output argument to be added.

```
CORBA::NVList_ptr arguments();
```

This method returns a pointer to an `NVList` object containing the arguments for this request. The pointer can be used to set or retrieve the argument values. For more information on `NVList`, see [“NVList” on page 66](#).

Caution You should never release the memory associated with this method's return value.

```
CORBA::ContextList_ptr contexts();
```

This method returns a pointer to a list of all the `Context` objects that are associated with this `Request`. For more information on the `Context` class, see [“Context” on page 16](#).

Caution You should never release the memory associated with this method's return value.

```
CORBA::Context_ptr ctx() const;
```

This method returns a pointer to the `Context` associated with this request.

```
void ctx(CORBA::Context_ptr ctx);
```

This method sets the `Context` to be used with this request. For more information on the `Context` class, see [“Context” on page 16](#).

Parameter	Description
<code>ctx</code>	The <code>Context</code> object to be associated with this request.

```
CORBA::Environment_ptr env();
```

This method returns a pointer to the `Environment` associated with this request. For more information on the `Environment` class, see [“Environment” on page 62](#).

```
CORBA::ExceptionList_ptr exceptions();
```

This method returns a pointer to a list of all the exceptions that this request may raise.

Caution You should never release the memory associated with this method's return value.

```
void get_response();
```

This method is used after the `send_deferred` method has been invoked to retrieve a response from the object implementation. If there is no response available, this method blocks the client application until a response is received.

```
void invoke();
```

This method invokes this `Request` on the ORB object associated with this request. This method blocks the client until a response is received from the object implementation. This `Request` should be initialized with the target object, operation name and arguments before this method is invoked.

```
const char* operation() const;
```

This method returns the name of the operation that this request performs.

```
CORBA::Boolean poll_response();
```

This non-blocking method is invoked after the `send_deferred` method to determine if a response has been received. This method returns `TRUE` if a response has been received, otherwise it returns `FALSE`.

```
CORBA::NamedValue_ptr result();
```

This method returns a pointer to a `NamedValue` object where the return value for the operation will be stored. The pointer can be used to retrieve the result value after the request has been processed by the object implementation. For more information on the `NamedValue` class, see “[NamedValue](#)” on page 65.

```
CORBA::Any& return_value();
```

This method returns a reference to an `Any` object that represents the return value of this `Request` object.

```
void set_return_type(CORBA::TypeCode_ptr tc);
```

This method sets the `TypeCode` of the return value that is expected. You must set the return value's type before using the `invoke` method or one of the `send` methods.

Parameter	Description
<code>tc</code>	The return value's type.

```
void send_deferred();
```

Like the `invoke` method, this method sends this `Request` to the object implementation. Unlike the `invoke` method, this method does not block waiting for a response. The client application can retrieve the response using the `get_response` method.

```
void send_oneway();
```

This method invokes this `Request` as a *oneway* operation. Oneway operations do not block and do not result in a response being sent from the object implementation to the client application.

```
CORBA::Object_ptr target() const;
```

This method returns a reference to the target object on which this request will operate.

```
static CORBA::Request_ptr _duplicate(CORBA::Request_ptr ptr);
```

This static method increments the reference count for the specified object and then returns a pointer to that object.

Parameter	Description
<code>ptr</code>	The object to be duplicated.

```
static CORBA::Request_ptr _nil();
```

This static method returns a `NULL` pointer that can be used to initialize a `CORBA::Request_ptr` object.

```
static void _release(CORBA::Request *ptr);
```

This static method decrements the reference count for the specified object. When the count reaches zero, all memory managed by the object is released and the object is deleted.

Parameter	Description
<code>ptr</code>	The object to be released.

ServerRequest

The `ServerRequest` class is used to represent an operation request received by an object implementation that is using the Dynamic Skeleton Interface. When the POA receives a client operation request, it invokes the object implementation's `invoke` method and passes an object of this type.

This class provides the methods needed by the object implementation to determine the operation being requested and the arguments. It also provides methods for setting the return value and reflecting exceptions to the client application.

You should never attempt to free memory associated with any value returned by this class.

Include file

The `corba.h` file should be included when you use this class.

ServerRequest methods

```
void arguments(CORBA::NVList_ptr param);
```

This method sets the parameter list for this request.

Parameter	Description
<code>params</code>	The parameter list to be filled in. You must initialize this list with the appropriate number of <code>Any</code> objects and set their type and flag values prior to invoking this method.

```
CORBA::Context_ptr ctx();
```

This method returns the `Context` object associated with the request.

Caution You should never release the memory associated with this method's return value.

```
void exception(CORBA::Any_ptr exception);
```

This method is used to reflect the specified exception to the client application.

Parameter	Description
<code>exception</code>	The exception that was raised. If this pointer is <code>NULL</code> , a <code>CORBA::UnknownUserException</code> is reflected.

```
const char *operation() const;
```

Returns the name of the operation being requested.


```
const char* op_name() const
```

This method returns the name of the operation associated with the request. The object implementation uses this name to determine if the request is valid, to perform the appropriate processing to fulfill the request, and to return the appropriate value to the client.

```
void params(CORBA::NVList_ptr params);
```

This method accepts an `NVList` object initialized with the appropriate number of `Any` objects. The method fills the `NVList` in with the parameters supplied by the client.

Parameter	Description
<code>params</code>	The parameter list to be filled in. You must initialize this list with the appropriate number of <code>Any</code> objects and set their type and flag values prior to invoking this method.

```
void result(CORBA::Any_ptr result);
```

This method sets the result that is to be reflected to the client application.

Parameter	Description
<code>result</code>	An <code>Any</code> object representing the return value.

```
void set_exception(const CORBA::Any& a);
```

This method sets the exception that is to be reflected to the client application.

Parameter	Description
<code>a</code>	An <code>Any</code> object representing the exception.

```
void set_result(const CORBA::Any& a);
```

This method sets the result that is to be reflected to the client application.

Parameter	Description
<code>a</code>	An <code>Any</code> object representing the return value.

```
static CORBA::ServerRequest_ptr _duplicate(CORBA::ServerRequest_ptr ptr);
```

This static method increments the reference count for the specified object and then returns a pointer to the object.

Parameter	Description
<code>ptr</code>	The object to be duplicated.

```
static CORBA::ServerRequest_ptr _nil();
```

This static method returns a `NULL` pointer that can be used for initialization purposes.

```
static void _release(CORBA::ServerRequest *ptr);
```

This static method decrements the reference count for the specified object. When the count reaches zero, all memory managed by the object is released and the object is deleted.

Parameter	Description
<code>ptr</code>	The object to be released.

TCKind

```
enum TCKind
```

This enumeration describes the various types that a `TypeCode` object, described in [“TypeCode” on page 75](#), may represent.

The values are shown in the following table.

Name	Meaning
<code>tk_abstract_interface</code>	abstract interface
<code>tk_alias</code>	alias
<code>tk_any</code>	Any
<code>tk_array</code>	array
<code>tk_boolean</code>	boolean
<code>tk_char</code>	char
<code>tk_double</code>	double
<code>tk_enum</code>	enum
<code>tk_except</code>	exception
<code>tk_fixed</code>	fixed type
<code>tk_float</code>	float
<code>tk_long</code>	long
<code>tk_longdouble</code>	long double
<code>tk_longlong</code>	long long
<code>tk_native</code>	native type
<code>tk_null</code>	NULL
<code>tk_objref</code>	object reference
<code>tk_octet</code>	octet string
<code>tk_Principal</code>	Principal
<code>tk_sequence</code>	sequence
<code>tk_short</code>	short
<code>tk_string</code>	string
<code>tk_struct</code>	struct
<code>tk_TypeCode</code>	TypeCode
<code>tk_ulonglong</code>	unsigned long long
<code>tk_union</code>	union
<code>tk_ulong</code>	unsigned long
<code>tk_ushort</code>	unsigned short
<code>tk_value</code>	value
<code>tk_value_box</code>	value box
<code>tk_void</code>	void
<code>tk_wchar</code>	Unicode character
<code>tk_wstring</code>	Unicode string

TypeCode

```
class CORBA::TypeCode
```

The `TypeCode` class represents the various types that can be defined in IDL. Type codes are most often used to define the type of value being stored in an `Any` object, described in “[Any methods](#)” on page 49. Type codes may also be passed as parameters to method invocations.

`TypeCode` objects can be created using the various `CORBA::ORB.create_<type>_tc` methods, whose description begins in the Core interfaces and classes, [Object](#) section. You may also use the constructors listed here.

Include file

Include the `corba.h` file when you use this class.

TypeCode constructors

```
CORBA::TypeCode(CORBA::TCKind kind, CORBA::Boolean is_constant);
```

This method constructs a `TypeCode` object for types that do not require any additional parameters. A `BAD_PARAM` exception is raised if `kind` is not a valid type for this constructor.

Parameter	Description
<code>kind</code>	Describes the type of object being represented. Must be one of the following: <code>CORBA::tk_null</code> , <code>CORBA::tk_void</code> , <code>CORBA::tk_short</code> , <code>CORBA::tk_long</code> , <code>CORBA::tk_ushort</code> , <code>CORBA::tk_ulong</code> , <code>CORBA::tk_float</code> , <code>CORBA::tk_double</code> , <code>CORBA::tk_boolean</code> , <code>CORBA::tk_char</code> , <code>CORBA::tk_octet</code> , <code>CORBA::tk_any</code> , <code>CORBA::tk_TypeCode</code> , <code>CORBA::tk_Principal</code> , <code>CORBA::tk_longlong</code> , <code>CORBA::tk_ulonglong</code> , <code>CORBA::tk_longdouble</code> , or <code>CORBA::tk_wchar</code> , <code>CORBA::tk_fixed</code> , <code>CORBA::tk_value</code> , <code>CORBA::tk_value_box</code> , <code>CORBA::native</code> , <code>CORBA::tk_abstract_interface</code> .
<code>is_constant</code>	If <code>TRUE</code> , the <code>TypeCode</code> object is to be considered a constant. Otherwise, the <code>TypeCode</code> object is not a constant.

TypeCode methods

```
CORBA::TypeCode_ptr content_type() const;
```

This method returns the `TypeCode` of the elements in a sequence or array. It also will return the type of an alias. A `BadKind` exception is raised if this object's kind is not `CORBA::tk_sequence`, `CORBA::tk_array`, or `CORBA::tk_alias`.

```
CORBA::Long default_index() const;
```

This method returns the default index of a `TypeCode` representing a union. If this object's kind is not `CORBA::tk_union`, a `BadKind` exception is raised.

```
CORBA::TypeCode_ptr discriminator_type() const;
```

This method returns the discriminator type of a `TypeCode` representing a union. If this object's kind is not `CORBA::tk_union`, a `BadKind` exception is raised.

```
>CORBA::Boolean equal(CORBA::TypeCode_ptr tc) const;
```

This method compares this object with the specified `TypeCode`. If they match in every respect, `TRUE` is returned. Otherwise, `FALSE` is returned.

Parameter	Description
<code>tc</code>	The object to be compared to this object.

```
const char* id() const;
```

This method returns the repository identifier of the type being represented by this object. If the type being represented does not have a repository identifier, a `BadKind` exception is raised. Types that have a repository identifier include:

- `CORBA::tk_struct`
- `CORBA::tk_union`
- `CORBA::tk_enum`
- `CORBA::tk_alias`
- `CORBA::tk_except`
- `CORBA::tk_objref`

```
CORBA::TCKind kind() const
```

This method returns this object's kind.

```
CORBA::ULong length() const;
```

This method returns the length of the string, sequence, or array represented by this object. The length of a string is the number of characters. The length of an array or sequence is the number of elements. A `BadKind` exception is raised if this object's kind is not `CORBA::tk_string`, `CORBA::tk_sequence`, or `CORBA::tk_array`.

```
CORBA::ULong member_count() const;
```

This method returns the member count of the type being represented by this `TypeCode` object. If the type being represented does not have members, a `BadKind` exception is raised. Types that have members include:

- `CORBA::tk_struct`
- `CORBA::tk_union`
- `CORBA::tk_enum`
- `CORBA::tk_except`

```
CORBA::Any_ptr member_label(CORBA::ULong index) const;
```

This method returns the label of the member with the specified `index` from a `TypeCode` object for a union. If this object's kind is not `CORBA::tk_union`, a `BadKind` exception is raised. If the index is invalid, a `Bounds` exception is raised.

Parameter	Description
<code>index</code>	The label of the union member whose type is to be returned. This index is zero-based.

```
const char *member_name(CORBA::ULong index) const;
```

This method returns the name of the member with the specified index from the type being represented by this object. If the type being represented does not have members, a `BadKind` exception is raised. If the index is invalid, a `Bounds` exception is raised.

Types that have members include:

- `CORBA::tk_struct`
- `CORBA::tk_union`
- `CORBA::tk_enum`
- `CORBA::tk_except`

Parameter	Description
<code>index</code>	The zero-based index of the member whose name is to be returned.

```
CORBA::TypeCode_ptr member_type(CORBA::ULong index) const;
```

This method returns the type of the member with the specified index from the type being represented by this object. If the type being represented does not have members with types, a `BadKind` exception is raised. If the index is invalid, a `Bounds` exception is raised. Types that have members include:

- `CORBA::tk_union`
- `CORBA::tk_except`

Parameter	Description
<code>index</code>	The zero-based index of the member whose name is to be returned.

```
const char *name() const;
```

This method returns the name of the type represented by this object. If the type does not have a name, a `BadKind` exception is raised. Types that have a name include:

- `CORBA::tk_objref`
- `CORBA::tk_struct`
- `CORBA::tk_union`
- `CORBA::tk_enum`
- `CORBA::tk_alias`
- `CORBA::tk_except`

```
static CORBA::TypeCode_ptr _duplicate(CORBA::TypeCode_ptr obj);
```

This static method duplicates the specified `TypeCode` object.

Parameter	Description
<code>obj</code>	The object to be duplicated.

```
static CORBA::TypeCode_ptr _nil();
```

This static method returns a `NULL` `TypeCode` pointer that can be used for initialization purposes.

```
static void _release(CORBA::TypeCode_ptr obj);
```

This static method decrements the reference count to the specified object. When the reference count is zero, it also frees all memory that it is managing and then deletes the object.

Parameter	Description
obj	The object to be released.

```
CORBA::Boolean equivalent (CORBA_TypeCode_ptr tc) const;
```

The `equivalent` operation is used by the ORB when determining the type equivalence for values stored in an IDL.

```
CORBA_TypeCode_ptr get_compact_typecode() const;
```

The `get_compact_code` operation strips out all optional name & member name fields, but it leaves all alias typecodes intact.

```
virtual CORBA::Visibility member_visibility(CORBA::ULong index) const;
```

This method returns the `Visibility` of the valuetype member identified by index.

Note The `member_visibility` operation can only be invoked on valuetype TypeCodes, not on valueboxes (or boxed values).

```
virtual CORBA::ValueModifier type_modifier() const;
```

The `type_modifier` operations can only be invoked on non-boxed valuetype TypeCodes. This method returns the `ValueModifier` that applies to the valuetype represented by the target TypeCode.

```
virtual CORBA::TypeCode_ptr concrete_base_types()
```

The `concrete_base_types` operations can only be invoked on non-boxed valuetype TypeCodes. If the value represented by the target TypeCode has a concrete base valuetype, this method returns a TypeCode for the concrete base, otherwise it returns a `nil` TypeCode reference.

Interface repository interfaces and classes

This section describes the classes and interfaces that you can use to access the interface repository. The interface repository maintains information on modules and the interfaces they contain as well as other types like operations, attributes, and constants.

AliasDef

```
class CORBA::AliasDef : public CORBA::TypedefDef
```

This class is derived from the `TypedefDef` class and represents an alias for a `typedef` that is stored in the interface repository. This class provides methods for setting and obtaining the `IDLType` of the original `typedef`.

For more information on the `TypedefDef` class, see “[TypedefDef](#)” on page 108. For more information on the `IDLType` class, see “[IDLType](#)” on page 95.

AliasDef methods

```
CORBA::IDLType original_type_def();
```

This method returns the `IDLType` of the original `typedef` for which this object is an alias.

```
void original_type_def(CORBA::IDLType_ptr val);
```

This method sets the `IDLType` of the original `typedef` for which this object is an alias.

Parameter	Description
<code>val</code>	The <code>IDLType</code> to set for this alias.

ArrayDef

```
class CORBA::ArrayDef : public CORBA::IDLType
```

This class is derived from the `IDLType` class and represents an array that is stored in the interface repository. It provides methods for setting and obtaining the type of the elements in the array as well as the length of the array.

ArrayDef methods

```
CORBA::TypeCode element_type();
```

This method returns the `TypeCode` of the array's elements.

```
CORBA::IDLType_ptr element_type_def();
```

This method returns the `IDLType` of the elements stored in this array.

```
void element_type_def(CORBA::IDLType_ptr element_type_def);
```

This method sets the `IDLType` of the elements stored in the array.

Parameter	Description
<code>element_type_def</code>	The <code>IDLType</code> of the elements in the array.

```
CORBA::ULong length();
```

This method returns the number of elements in the array.

```
void length(CORBA::ULong length);
```

This method sets the number of elements in the array.

Parameter	Description
<code>length</code>	The number of elements in the array.

AttributeDef

```
class CORBA::AttributeDef : public CORBA::Contained, public CORBA::Object
```

The class is used to represent an interface attribute that is stored in the interface repository. It provides methods for setting and obtaining the attribute's mode, `typedef`. A method is also provided for obtaining the attribute's type.

AttributeDef methods

```
CORBA::AttributeMode mode();
```

This method returns the mode of the attribute. The return value will be either `CORBA::AttributeMode ATTR_READONLY` for read only attributes or `CORBA::AttributeMode ATTR_NORMAL` for read-write ones. See [“AttributeMode” on page 82](#) for more information.


```
void mode(CORBA::AttributeMode _val);
```

This method sets the mode of the attribute.

Parameter	Description
<code>_val</code>	The mode to set.

```
CORBA::TypeCode_ptr type();
```

This method returns the `TypeCode` that represents the attribute's type.

```
CORBA::IDLType_ptr type_def();
```

This method returns this object's `IDLType`.

```
void type_def(CORBA::IDLType_ptr type_def);
```

This method sets the `IDLType` for which this object.

Parameter	Description
<code>type_def</code>	The <code>IDLType</code> of this object.

AttributeDescription

```
struct CORBA::AttributeDescription
```

The `AttributeDescription` structure describes an attribute that is stored in the interface repository.

AttributeDescription members

```
CORBA::Identifier_var name
```

The name of the attribute.

```
CORBA::RepositoryId_var id
```

The repository id of the attribute.

```
CORBA::RepositoryId_var defined_in
```

The repository id of the interface in which this attribute is defined.

```
CORBA::String_var version
```

The attribute's version.

```
CORBA::TypeCode_var type
```

The attribute's IDL type.

```
CORBA::AttributeMode mode
```

The mode of this attribute.

AttributeMode

enum CORBA::AttributeMode

The enumeration defines the values used to represent the mode of an attribute; either read-only or normal (read-write).

AttributeMode values

Constant	Represents
ATTR_NORMAL	This is a read-write attribute.
ATTR_READONLY	This is a read-only attribute.

ConstantDef

class CORBA::ConstantDef : public CORBA::Contained

The class is used to represent a constant definition that is stored in the interface repository. This interface provides methods for setting and obtaining the constant's type, value, and typedef.

ConstantDef methods

CORBA::TypeCode_ptr type();

This method returns the TypeCode representing the object's type.

CORBA::IDLType_ptr type_def();

This method returns this object's IDLType.

void type_def(CORBA::IDLType_ptr type_def);

This method sets the IDLType of the constant.

Parameter	Description
type_def	The IDLType of this constant.

CORBA::Any *value();

This method returns a pointer to an Any object representing this object's value.

void value(CORBA::Any& _val);

This method sets the value of this constant.

Parameter	Description
_val	An Any object that represents this object's value.

ConstantDescription

```
struct CORBA::ClassName
```

The `ConstantDescription` structure describes a constant that is stored in the interface repository.

ConstantDescription members

```
CORBA::Identifier_var name
```

The name of the constant.

```
CORBA::RepositoryId_var id
```

The repository id of the constant.

```
CORBA::RepositoryId_var defined_in
```

The name of the module or interface in which this constant is defined.

```
CORBA::String_var version
```

The constant's version.

```
CORBA::TypeCode_var type
```

The constant's IDL type.

```
CORBA::Any value
```

The value of this constant.

Contained

```
class CORBA::Contained : public CORBA::IObject, public CORBA::Object
```

The `Contained` class is used to derive all interface repository objects that are themselves contained within another interface repository object. This class provides methods for:

- Setting and retrieving the object's name and version.
- Determining the `Container` that contains this object.
- Obtaining the object's absolute name, containing repository, and description.
- Moving an object from one container to another.

Include file

Include the files **corba.h** and **ir_c.hh** when you use this class.

```
interface Contained: IObject {
    attribute RepositoryId id;
    attribute Identifier name;
    attribute String_var version;

    readonly attribute Container defined_in;
    readonly attribute ScopedName absolute_name;
    readonly attribute Repository containing_Repository;
    struct Description {
        DefinitionKind kind;
        any value;
    };
    Description describe();
    void move(
        in Container new_Container,
        in Identifier new_name,
        in String_var new_version
    );
};
```

Contained methods

CORBA::String_var **absolute_name()**;

This method returns the absolute name, which is the name that uniquely identifies this object within its containing Repository. If the object's `defined_in` attribute (set when the object is created) references a Repository, then the absolute name is simply the object's name preceded by the string "::".

CORBA::Repository_ptr **containing_repository()**;

Returns a pointer to the repository that contains this object.

CORBA::Container_ptr **defined_in()**;

Returns a pointer to the Container where this object is defined.

Description* **describe()**;

Returns this object's Description. See ["Description" on page 91](#) for more information on the Description structure.

CORBA::String_var **id()**;

Returns this object's repository identifier.

void **id(const char *id)**;

Sets the repository identifier that uniquely identifies this object.

Parameter	Description
id	The repository identifier for this object.

```
CORBA::String_var name();
```

This method returns the name which uniquely identifies the object within the scope of its container.

```
void name(const char * val);
```

This method sets the name of the contained object.

Parameter	Description
name	The object's name.

```
CORBA::String_var version();>
```

This method returns the object's version. The version distinguishes this object from other objects that have the same name.

```
void version(CORBA::String_var& val);
```

This method sets this object's version.

Parameter	Description
val	The object's version.

```
void move(CORBA::Container_ptr new_container, const char *new_name,
CORBA::String_var& new_version);
```

Moves this object from its current Container to the new_container.

Parameter	Description
new_container	The Container to which this object is being moved.
new_name	The new name for the object.
new_version	The new version specification for the object.

Container

```
class CORBA::Container : public CORBA::Container, public CORBA::Object
```

The Container class is used to create a containment hierarchy in the interface repository. A Container object holds object definitions derived from the Contained class. All object definitions derived from the Container class, with the exception of the Repository class, also inherit from the Contained class.

The Container provides methods to create types of IDL types defined in **orbtypes.h**, including InterfaceDef, ModuleDef and ConstantDef classes, but not the ValueMemberDef class. The defined_in attribute of each definition that is created is initialized to point to this object.

Include file

The `corba.h` and `ir_c.hh` files should be included when you use this class.

```
interface Container: IRObject {
    Contained lookup(in ScopedName search_name);
    ContainedSeq contents(
        in DefinitionKind limit_type,
        in boolean exclude_inherited
    );
    ContainedSeq lookup_name(
        in Identifier search_name,
        in long levels_to_search,
        in CORBA::DefinitionKind limit_type,
        in boolean exclude_inherited
    );
    struct Description {
        Contained Contained_object;
        DefinitionKind kind;
        any value;
    };
    typedef sequence<Description> DescriptionSeq;
    DescriptionSeq describe_contents(
        in DefinitionKind limit_type,
        in boolean exclude_inherited,
        in long max_returned_objs
    );
};
```

Container methods

`CORBA::AbstractInterfaceDef_ptr` **create_abstract_interface**(const char* `_arg_id`, const char* `_arg_name`, const char* `_arg_name`, const char* `_arg_version`, const `CORBA::AbstractInterfaceDefSeq&` `_arg_base_interfaces`)

This method creates an `AbstractInterfaceDef` object with the specified attributes in the Container and returns a pointer to the newly created object.

Parameter	Description
<code>id</code>	The interface id.
<code>name</code>	The interface name.
<code>version</code>	The interface version.
<code>base_interfaces</code>	A list of all abstract interfaces from which this interface inherits.

`CORBA::ContainedSeq *` **contents**(`CORBA::DefinitionKind` `limit_type`, `CORBA::Boolean` `exclude_inherited`);

This method returns the list of definitions of contained objects that are either directly contained or inherited into the container. You can use this method to navigate through the hierarchy of object definitions in the `Repository`. This method returns all object definitions contained by modules in the `Repository`, followed by all object definitions contained within each of those modules.

Parameter	Description
<code>limit_type</code>	The interface object types to be returned. If you specify <code>dk_all</code> , objects of all types are returned.
<code>exclude_inherited</code>	If set to <code>TRUE</code> , inherited objects are not returned.

```
CORBA::AliasDef_ptr create_alias(const char * id, const char *name, const
CORBA::String_var& version, CORBA::IDLType_ptr original_type);
```

This method creates an `AliasDef` object with the specified attributes in this `Container` and returns a pointer to the newly created object.

Parameter	Description
<code>id</code>	The alias's id.
<code>name</code>	The alias's name.
<code>version</code>	The alias's version.
<code>original_type</code>	The type of the object for which this object is an alias.

```
CORBA::ConstantDef_ptr create_constant(const char * id, const char *name, const
CORBA::String_var& version, CORBA::IDLType_ptr type, const CORBA::Any& value);
```

This method creates a `ConstantDef` object with the specified attributes in this `Container` and returns a pointer to the newly created object.

Parameter	Description
<code>id</code>	The constant's id.
<code>name</code>	The constant's name.
<code>version</code>	The constant's version.
<code>type</code>	The type of the value specified below.
<code>value</code>	The constant's value.

```
CORBA::EnumDef_ptr create_enum(const char * id, const char *name, const
CORBA::String_var& version, const CORBA::EnumMemberSeq& members);
```

This method creates an `EnumDef` object with the specified attributes in this `Container` and returns a pointer to the newly created object.

Parameter	Description
<code>id</code>	The enumeration's id.
<code>name</code>	The enumeration's name.
<code>version</code>	The enumeration's version.
<code>members</code>	A list of the enumeration's fields.

```
CORBA::ExceptionDef_ptr create_exception(const char * id, const char *name,
const CORBA::String_var& version, const CORBA::StructMemberSeq& members);
```

This method creates an `ExceptionDef` object with the specified attributes in this `Container` and returns a pointer to the newly created object.

Parameter	Description
<code>id</code>	The exception's id.
<code>name</code>	The exception's name.
<code>version</code>	The exception's version.
<code>members</code>	The sequence for the structure's fields, if any.

```
CORBA::InterfaceDef_ptr create_interface(const char * id, const char *name, const
CORBA::String_var& version, const CORBA::InterfaceDefSeq& base_interfaces);
```

This method creates an `InterfaceDef` object with the specified attributes in this Container and returns a pointer to the newly created object.

Parameter	Description
<code>id</code>	The interface's id.
<code>name</code>	The interface's name.
<code>version</code>	The interface's version.
<code>base_interfaces</code>	A list of all interfaces that this interface inherits from.

```
CORBA::ModuleDef_ptr create_module(const char * id, const char *name,
const CORBA::String_var& version);
```

This method creates a `ModuleDef` object with the specified attributes in this Container and returns a pointer to the newly created object.

Parameter	Description
<code>id</code>	The module's id.
<code>name</code>	The module's name.
<code>version</code>	The module's version.

```
CORBA::StructDef_ptr create_struct(const char * id, const char *name, const
CORBA::String_var& version, const CORBA::StructMemberSeq& members);
```

This method creates a `StructureDef` object with the specified attributes in this Container and returns a pointer to the newly created object.

Parameter	Description
<code>id</code>	The structure's id.
<code>name</code>	The structure's name.
<code>version</code>	The structure's version.
<code>members</code>	The sequence for the structure's fields.

```
CORBA::UnionDef_ptr create_union(const char * id, const char *name, const
CORBA::String_var& version, CORBA::IDLType_ptr discriminator_type, const
CORBA::UnionMemberSeq& members);
```

This method creates a `UnionDef` object with the specified attributes in this Container and returns a pointer to the newly created object.

Parameter	Description
<code>id</code>	The Union's id.
<code>name</code>	The Union's name.
<code>version</code>	The Union's version.
<code>discriminator_type</code>	The type of the Union's discriminant value.
<code>members</code>	The sequence of each of the Union's fields.


```
CORBA::DescriptionSeq * describe_contents(CORBA::DefinitionKind limit_type,
CORBA::Boolean exclude_inherited, CORBA::Long max_returned_objs);
```

This method returns a description for all definitions directly contained by or inherited into this container.

Parameter	Description
limit_type	The interface object types whose descriptions are to be returned. Specifying <code>dk_all</code> will return the descriptions for objects of all types.
exclude_inherited	If set to true, descriptions for inherited objects are not returned.
max_returned_objs	The maximum number of descriptions to be returned. If you set this parameter to -1, all objects are returned.

```
CORBA::Contained_ptr lookup(const char *search_name);
```

This method locates a definition relative to this container, given a scoped name. An absolute scoped name, one beginning with “:”, may be specified to locate a definition within the enclosing repository. If no object is found, a `NULL` value is returned.

Parameter	Description
search_name	The object's interface name.

```
CORBA::ContainedSeq * lookup_name(const char *search_name, CORBA::Long
levels_to_search, CORBA::DefinitionKind limit_type, CORBA::Boolean
exclude_inherited);
```

This method locates an object by name within a particular object. The search can be constrained by the number of levels in the hierarchy to be searched, the object type, and whether inherited objects should be returned.

Parameter	Description
search_name	The contained object's name.
levels_to_search	The number of levels in the hierarchy to search. If you set this parameter to a value of -1, all levels are searched. If you set this parameter to 1, only this object is searched.
limit_type	The interface object types to be returned. Specifying <code>dk_all</code> will return objects of all types.
exclude_inherited	If set to true, inherited objects are not returned.

```
CORBA::ValueDef_ptr create_value(const char * id, const char *name, const char
version, CORBA::boolean is_custom, CORBA::boolean is_abstract, const
CORBA::ValueDef_ptr _base_value, CORBA::boolean is_truncatable, const
CORBA::ValueDefSeq& abstract_base_values, const CORBA::InterfaceDefSeq& supported
_interfaces, const CORBA::InitializerSeq& initializers)
```

This method creates a `ValueDef` object with the specified attributes in this `Container` and returns a reference to the newly created object.

Parameter	Description
id	The structure's repository id.
name	The structure's name.
version	The structure's version.
is_custom	If set to true, creates a custom valuetype.
is_abstract	If set to true, creates and abstract valuetype.
base_values	The list of supported base values.

Parameter	Description
is_truncatable	If set to true, creates a truncatable valuetype.
abstract_base_values	The list of supported abstract base values.
supported_interfaces	The list of supported interfaces.
initializer	The list of initializers this value type supports

`CORBA::ValueBoxDef_ptr create_value_box(const char* id, const char* name, const char* version, CORBA::IDLType_ptr original_type)`

This method creates a `ValueBoxDef` object in this `Container` with the specified attributes and returns a reference to the newly created object.

Parameter	Description
id	The structure's repository id.
name	The structure's name.
version	The structure's version.
original_type	The IDL type of the original object for which this is an alias.

DefinitionKind

enum `CORBA::DefinitionKind`

The constants in the `DefinitionKind` enumeration define the possible types of interface repository objects.

DefinitionKind values

Constant	Represents
<code>dk_none</code>	Exclude all types (used in repository lookup methods)
<code>dk_all</code>	All possible types (used in repository lookup methods)
<code>dk_Alias</code>	Alias
<code>dk_Array</code>	Array
<code>dk_Attribute</code>	Alias
<code>dk_Constant</code>	Constant
<code>dk_Enum</code>	Enum
<code>dk_Exception</code>	Exception
<code>dk_Fixed</code>	Fixed
<code>dk_Interface</code>	Interface
<code>dk_Module</code>	Module
<code>dk_Native</code>	Native
<code>dk_Operation</code>	Interface Operation
<code>dk_Primitive</code>	Primitive type (such as int or long)
<code>dk_Repository</code>	Repository
<code>dk_Sequence</code>	Sequence
<code>dk_String</code>	String
<code>dk_Struct</code>	Struct
<code>dk_Typedef</code>	Typedef
<code>dk_Union</code>	Union

Constant	Represents
dk_Value	ValueType
dk_ValueBox	ValueBox
dk_ValueMember	ValueMember
dk_Wstring	Unicode string

Description

```
struct CORBA::Container::Description
```

This structure provides a generic description for items in the interface repository that are derived from the `Contained` class.

Description members

```
CORBA::Contained_var contained_object
```

The object contained in this struct.

```
CORBA::DefinitionKind kind
```

The object's kind.

```
CORBA::Any value
```

The object's value.

EnumDef

```
class CORBA::EnumDef : public CORBA::TypedDef, public CORBA::Object
```

The class is used to describe an enumeration stored in the interface repository. This interface provides methods for setting and retrieving the enumeration's list of members.

EnumDef methods

```
CORBA::EnumMemberSeq *members();
```

This method returns the enumeration's list of members.

```
void members(CORBA::EnumMemberSeq members);
```

This method sets the enumeration's list of members.

Parameter	Description
members	The list of members.

ExceptionDef

```
class ExceptionDef : public CORBA::Contained
```

The class is used to describe an exception that is stored in the interface repository. This class provides methods for setting and retrieving the exception's list of members as well as a method for retrieving the exception's `TypeCode`.

ExceptionDef methods

```
CORBA::StructMemberSeq *members();
```

This method returns this exception's list of members.

```
void members(CORBA::StructMemberSeq& members);
```

This method sets the exception's list of members.

Parameter	Description
members	The list of members.

```
CORBA::TypeCode_ptr type();
```

This method returns the `TypeCode` that represents this exception's type.

ExceptionDescription

```
struct CORBA::ExceptionDescription
```

This structure is used to describe an exception that is stored in the interface repository.

ExceptionDescription members

```
CORBA::String_var defined_in
```

The repository Id of the module or interface in which this exception is defined.

```
CORBA::String_var id
```

The repository id of the exception.

```
CORBA::String_var name
```

The name of the exception.

```
CORBA::TypeCode_var type
```

The exception's IDL type.

```
CORBA::String_var version
```

The exception's version.

FixedDef

```
CORBA::FixedDef public CORBA::IDLType, public CORBA::Object
```

This interface is used to describe a fixed definition that is stored in the Interface Repository.

Methods

```
CORBA::UShort digits();
```

This method sets the number of digits for the fixed type.

```
void digits(CORBA::UShort _digits);
```

This method sets the attribute for fixed type.

```
CORBA::Short scale();
```

This method sets the scale for the fixed type.

```
void scale(CORBA::Short _scale);
```

This method sets the attribute for the fixed type.

FullInterfaceDescription

```
struct CORBA::FullInterfaceDescription
```

The FullInterfaceDescription structure describes an interface that is stored in the interface repository.

FullInterfaceDescription members

```
CORBA::String_var Name
```

The name of the interface.

```
CORBA::String_var id
```

The repository id of the interface.

```
CORBA::String_var defined_in
```

The name of the module or interface in which this interface is defined.

```
CORBA::String_var version
```

The interface's version.

```
CORBA::OpDescriptionSeq operations
```

The list of operations supported by this interface.

```
CORBA::AttrDescriptionSeq attributes
```

The list of attributes contained in this interface.

CORBA::RepositoryIdSeq **base_interfaces**

The interfaces from which this interface inherits.

CORBA::RepositoryIdSeq **derived_interfaces**

The interfaces derived from this interface.

CORBA::TypeCode_var **type**

This interface's TypeCode.

CORBA::Boolean **is_abstract**

Indicates whether or not this interface is abstract.

FullValueDescription

struct CORBA::FullValueDescription

This structure is used to represent a full value definition that is stored in the Interface Repository.

Variables

CORBA::String_var **name**

The name of the valuetype.

CORBA::String_var **id**

The repository id of the valuetype.

CORBA::Boolean **is_abstract**

If this variable is `true`, specifies an abstract valuetype.

CORBA::Boolean **is_custom**

If this variable is `true`, specifies custom marshalling for the valuetype.

CORBA::String_var **defined_in**

The repository id of the module in which this valuetype is defined.

CORBA::String_var **version**

The valuetype's version.

CORBA::OpDescriptionSeq **operations**

The list of operations offered by the valuetype.

CORBA::AttrDescriptionSeq **attributes**

The valuetype's list of valuetype's member attributes.

CORBA::ValueMemberSeq **members**

The array of value definitions.

CORBA::InitializerSeq **initializers**

The array of initializers.

CORBA::RepositoryIdSeq **supported_interfaces;**

The list of supported interfaces.

CORBA::RepositoryIdSeq **abstract_base_values;**

The list of abstract value types from which this valuetype inherits.

CORBA::Boolean **is_truncatable;**

If this variable is set to `true`, the value can be truncated to its base valuetype safely.

CORBA::String_var **base_values;**

The description of the value type from which this valuetype inherits.

CORBA::TypeCode_var **type**

The valuetype's IDL type code.

IDLType

```
class CORBA::IDLType : public CORBA::IObject, public CORBA::Object
```

The `IDLType` class provides an abstract interface that is inherited by all interface repository definitions that represent IDL types. This class provides a method for returning an object's `Typecode`, which identifies the object's type. The `IDLType` is unique; the `Typecode` is not.

Include file

You should include the files `corba.h` and `ir_c.hh` when using this class.

```
interface IDLType:IObject {
    readonly attribute TypeCode type;
};
```

IDLType methods

```
CORBA::Typecode_ptr type();
```

This method returns the typecode of the current `IObject`.

InterfaceDef

```
class CORBA::InterfaceDef : public CORBA::Container, public CORBA::Contained,
public CORBA::IDLType
```

The `InterfaceDef` class is used to define an ORB object's interface that is stored in the interface repository.

For more information, see [“Container” on page 85](#), [“Contained” on page 83](#), and [“IDLType” on page 95](#).

Include file

You should include the files `corba.h` and `ir_c.hh` when you use this class.

```
interface InterfaceDef: Container, Contained, IDLType {
typedef sequence<RepositoryId> RepositoryIdSeq;
typedef sequence<OperationDescription> OpDescriptionSeq;
typedef sequence<AttributeDescription> AttrDescriptionSeq;
    attribute InterfaceDefSeq base_interfaces;
    attribute boolean is_abstract;
    readonly attribute InterfaceDefSeq
        derived_interfaces
boolean is_a(in RepositoryId interface_id);
    struct FullInterfaceDescription {
Identifier name;
RepositoryId id;
RepositoryId defined_in;
String_var version;
OpDescriptionSeq operations;
AttrDescriptionSeq attributes;
RepositoryIdSeq base_interfaces;
    RepositoryIdSeq derived_interfaces;
TypeCode type;
    boolean is_abstract;
};
FullInterfaceDescription describe_interface();
AttributeDef create_attribute(
    in RepositoryId id,
    in Identifier name,
    in String_var version,
    in IDLType type,
    in CORBA::AttributeMode mode
);
OperationDef create_operation(
    in RepositoryId id,
    in Identifier name,
    in String_var version,
    in IDLType result,
    in OperationMode mode,
    in ParDescriptionSeq params,
    in ExceptionDefSeq exceptions,
    in ContextIdSeq contexts
);
```



```

struct InterfaceDescription {
    Identifier name;
    RepositoryId id;
    RepositoryId defined_in;
    String_var version;
    RepositoryIdSeq base_interfaces;
    boolean is_abstract;
};
};

```

InterfaceDef methods

```
CORBA::InterfaceDefSeq *base_interfaces();
```

This method returns a list of interfaces from which this class inherits.

```
void base_interfaces(const CORBA::InterfaceDefSeq& val);
```

This method sets the list of the interfaces from which this class inherits.

Parameter	Description
val	The list of interfaces from which this interface inherits.

```
CORBA::AttributeDef_ptr create_attribute(const char * id, const char * name, const
CORBA::String_var& version, CORBA::IDLType_ptr type, CORBA::AttributeMode mode);
```

This method returns a pointer to a newly created `AttributeDef` that is contained in this object. The `id`, `name`, `version`, `type`, and `mode` are set to the values specified.

Parameter	Description
id	The interface id to use.
name	The interface name to use.
version	The interface version to use.
mode	The interface mode. See “ AttributeMode ” on page 82 for a list of possible values.

```
CORBA::OperationDef_ptr create_operation(const char *id, const char *name,
CORBA::String_var& version, CORBA::IDLType_ptr result, CORBA::OperationMode mode,
const CORBA::ParDescriptionSeq& params, const CORBA::ExceptionDefSeq& exceptions,
const CORBA::ContextIdSeq& contexts);
```

This method creates a new `OperationDef` that is contained by this object using the specified parameters. The `defined_in` attribute of the newly created `OperationDef` is set to identify this `InterfaceDef`.

Parameter	Description
id	The interface id for this operation.
name	The name of this operation.
version	The operation's version.
result	The IDL type returned by the operation.
mode	The mode of this operation—one-way or normal.
params	The list of parameters to pass to this operation.
exceptions	The list of exceptions raised by this operation.
contexts	Context lists are names of values expected in context and passed along with the request.

```
CORBA::InterfaceDef::FullInterfaceDescription *describe_interface();
```

This method returns the `FullInterfaceDescription` which describes this object's interface.

```
CORBA::Boolean is_a(const char * interface_id);
```

This method returns `true` if this interface is identical to or inherits from the specified interface directly or indirectly.

Parameter	Description
<code>interface_id</code>	The id of the interface to be checked against this interface.

InterfaceDescription

```
struct CORBA:: InterfaceDescription
```

This structure describes an object that is stored in the interface repository.

InterfaceDescription members

```
CORBA::String_var name
```

The name of the interface.

```
CORBA::String_var id
```

The interface's repository identifier.

```
CORBA::String_var defined_in
```

The name of the repository Id in which the interface is defined.

```
CORBA::String_var version
```

The interface's version.

```
CORBA::RepositoryIdSeq base_interfaces
```

A list of base interfaces for this interface.

```
CORBA::Boolean is_abstract
```

Indicates whether or not this interface is abstract.

IObject

```
class IObject : CORBA::Object
```

The `IObject` class offers the most generic interface for interface repository objects. The `Container` class, `IDLType`, `Contained`, and others are derived from this class.

Include file

You should include the files `corba.h` and `ir_c.hh` when you use this class.

```
interface IObject {
    readonly attribute DefinitionKind def_kind;
    void destroy();
};
```

IObject methods

```
CORBA::DefinitionKind def_kind();
```

This method returns the type of this interface repository object. See “[DefinitionKind](#)” on page 90 for a list of possible types.

```
void destroy();
```

This method deletes this object from the interface repository. If this object is a `Container`, this method also deletes all of its contents. If the object is currently contained by another object, it is removed. The `destroy` method returns the `Exception(CORBA::BAD_PARAM)` when invoked on a `PrimitiveDef` or `Repository` object. The `Repository` class is described in “[Repository](#)” on page 105 .

ModuleDef

```
class ModuleDef : CORBA::Container, CORBA::Contained
```

The class is used to represent an IDL module in the interface repository.

ModuleDescription

```
struct ModuleDescription
```

The `ModuleDescription` structure describes a module that is stored in the interface repository.

ModuleDescription members

```
CORBA::String_var name
```

The name of the module.

```
CORBA::String_var id
```

The repository id of the module.

CORBA::String_var **defined_in**

The name of the repository Id in which this module is defined.

CORBA::String_var **version**

The module's version.

NativeDef

class CORBA::NativeDef

This interface is used to represent a native definition that is stored in the Interface Repository.

OperationDef

class CORBA::OperationDef : public virtual CORBA::Contained, public CORBA::Object

The `OperationDef` class contains information about an interface operation that is stored in the interface repository. This class is derived from the `Contained` class, which is described in [“Contained” on page 83](#). The inherited `describe` method returns a `OperationDescription` structure that provides complete information on the operation.

Include file

You should include the files `corba.h` and `ir_c.hh` when you use this class.

```
interface OperationDef: Contained {
    typedef sequence<ParameterDescription> ParDescriptionSeq;
    typedef Identifier ContextIdentifier;
    typedef sequence<ContextIdentifier> ContextIdSeq;
    typedef sequence<ExceptionDef> ExceptionDefSeq;
    typedef sequence<ExceptionDescription> ExcDescriptionSeq;
    readonly attribute TypeCode result;
    attribute IDLType result_def;
    attribute ParDescriptionSeq params;
    attribute CORBA::OperationMode mode;
    attribute ContextIdSeq contexts;
    attribute ExceptionDefSeq exceptions;

    readonly attribute OperationKind bind;
};
struct OperationDescription {
    Identifier name;
    RepositoryId id;
    RepositoryId defined_in;
    String_var version;
    TypeCode result;
    OperationMode mode;
    ContextIdSeq contexts;
    ParDescriptionSeq parameters;
    ExcDescriptionSeq exceptions;
};
```

OperationDef methods

```
CORBA::ContextIdSeq * contexts();
```

This method returns the list of context identifiers that apply to the operation.

```
void context(const CORBA::ContextIdSeq& val);
```

This method sets the list of context identifiers that apply to this operation.

Parameter	Description
val	The list of context identifiers.

```
CORBA::ExceptionDefSeq * exceptions();
```

This method returns the list of the exception types that can be raised by this operation.

```
void exceptions(const CORBA::ExceptionDefSeq& val);
```

This method sets the list of exception types that may be raised by this operation.

Parameter	Description
val	The list of exceptions that this operation may raise.

```
CORBA::OperationMode mode();
```

This method returns the mode of the operation represented by this `OperationDef`. The mode may be normal or one-way. Operations that have a normal mode are synchronous and return a value to the client application. One-way operations do not block and no response is sent from the object implementation to the client.

```
void mode(CORBA::OperationMode val);
```

This method sets the mode of the operation.

Parameter	Description
val	The desired mode of this operation, either <code>OP_ONEWAY</code> or <code>OP_NORMAL</code> . Go to the OperationMode section for more information.

```
CORBA::ParDescriptionSeq * params();
```

This method returns a pointer to a list of `ParameterDescription` structures that describe the parameters to this `OperationDef`.

```
void params(const CORBA::ParDescriptionSeq& val);
```

This method sets the list of the `ParameterDescription` structures for this `OperationDef`. The order of the structures is significant and should correspond to the order defined in the IDL definition for the operation.

Parameter	Description
val	The list of <code>ParameterDescription</code> structures.

```
CORBA::TypeCode_ptr result();
```

This method returns a pointer to a `TypeCode` representing the type of the value returned by this `Operation`. The `TypeCode` is a read-only attribute.

```
CORBA::IDLType_ptr result_def();
```

This method returns a pointer to the definition of the IDL type returned by this `OperationDef`.

```
void result_def(CORBA::IDLType_ptr val);
```

This method sets the definition of the type returned by this `OperationDef`.

Parameter	Description
<code>val</code>	A pointer to the type definition to use.

OperationDescription

```
struct CORBA::OperationDescription
```

The `OperationDescription` structure describes an operation that is stored in the interface repository.

OperationDescription members

```
CORBA::String_var name
```

The name of the operation.

```
CORBA::String_var id
```

The repository id of the operation.

```
CORBA::String_var defined_in
```

The repository id of the interface or valuetype in which this operation is defined.

```
CORBA::String_var version
```

The operation's version.

```
CORBA::TypeCode_var result
```

The operation's result.

```
CORBA::OperationMode mode
```

The operation's mode.

```
CORBA::ContextIdSeq contexts
```

The operation's associated context list.

```
CORBA::ParameterDescriptionSeq parameters
```

The operation's parameters.

```
CORBA::ExceptionDescriptionSeq exceptions
```

The exceptions that this operation may raise.

OperationMode

enum CORBA:**OperationMode**

The enumeration defines the values used to represent the mode of an operation; either one-way or normal. One-way operations are those for which the client application does not expect a response. Normal requests involve a response being sent to the client by the object implementation that contains the results of the request.

OperationMode values

Constant	Represents
OP_NORMAL	A normal operation request.
OP_ONeway	A one-way operation request.

ParameterDescription

struct CORBA::**ParameterDescription**

The `ParameterDescription` structure describes a parameter for an operation that is stored in the interface repository.

ParameterDescription members

CORBA::`String_var` **name**

The name of the parameter.

CORBA::`TypeCode_var` **type**

The parameter's typecode.

CORBA::`IDLType_var` **type_def**

The parameter's IDL type.

CORBA::`ParameterMode` **mode**

The parameter's mode.

ParameterMode

enum CORBA::**ParameterMode**

The values that represent the possible modes of parameters to operations.

ParameterMode values

Constant	Represents
PARAM_IN	Parameter is for input from the client to the server.
PARAM_OUT	Parameter is for output of results from the server to the client.
PARAM_INOUT	Parameter may be used for both input from the client and output from the server.

PrimitiveDef

```
class PrimitiveDef : public CORBA::IDLType, public CORBA::Object
```

The class is used to describe a primitive (such as an `int` or a `long`) that is stored in the interface repository. It provides a method for retrieving what kind of primitive it is.

PrimitiveDef methods

```
CORBA::PrimitiveKind kind();
```

This method returns the kind of primitive represented by this object.

PrimitiveKind

```
enum CORBA::PrimitiveKind
```

The `PrimitiveKind` enumeration contains the constants that define the primitive types of objects that may be stored in the interface repository.

PrimitiveKind values

Constant	Represents
pk_null	Null value
pk_void	Void
pk_short	Short
pk_long	Long
pk_ushort	Unsigned short
pk_ulong	Unsigned long
pk_float	Float
pk_double	Double
pk_boolean	Boolean
pk_char	Character
pk_octet	Octet
pk_any	Any
pk_TypeCode	TypeCode
pk_Principal	Principal
pk_string	String
pk_objref	Object reference

Constant	Represents
pk_longlong	Long long
pk_ulonglong	Unsigned long long
pk_longdouble	Long double
pk_wchar	Unicode character
pk_wstring	Unicode string

Repository

```
class Repository : public CORBA::Container, public CORBA::Object
```

The `Repository` class provides access to the interface repository and is derived from the `Container` class. See [“Container” on page 85](#) for more information.

Include file

You should include the files `corba.h` and `ir_c.hh` when using this class.

```
interface Repository: Container {
    Contained lookup_id(in RepositoryId search_id);
    PrimitiveDef get_primitive(in CORBA::PrimitiveKind kind);
    StringDef create_string(in unsigned long bound);
    WStringDef create_wstring(in unsigned long bound)
    SequenceDef create_sequence(
        in unsigned long bound,
        in IDLType element_type
    );
    ArrayDef create_array(
        in unsigned long length,
        in IDLType element_type
    );
    FixedDef create_fixed(
        in unsigned short digits,
        in short scale
    );
};
```

Repository methods

```
CORBA::ArrayDef_ptr create_array(CORBA::ULong length, CORBA::IDLType_ptr element_type);
```

This method creates a new `ArrayDef` and returns a pointer to it.

Parameter	Description
<code>length</code>	The maximum number of elements in the array. This value must be greater than zero.
<code>element_type</code>	The IDLType of the elements in the array.

```
CORBA::SequenceDef_ptr create_sequence(CORBA::Ulong bound, CORBA::IDLType_ptr element_type);
```

This method creates a new `SequenceDef` object and returns a pointer to it.

Parameter	Description
<code>bound</code>	The maximum number of items in the sequence. This value must be greater than zero.
<code>element_type</code>	A pointer to the <code>IDLType</code> of the items in the sequence.

```
CORBA::StringDef_ptr create_string(CORBA::Ulong bound);
```

This method creates a new `StringDef` object and returns a pointer to it.

Parameter	Description
<code>bound</code>	The maximum length of the string. This value must be greater than zero.

```
CORBA::WstringDef_ptr create_wstring(CORBA::Ulong bound);
```

This method creates a new `WstringDef` object and returns a pointer to it.

Parameter	Description
<code>bound</code>	The maximum length of the string. This value must be greater than zero.

```
CORBA::PrimitiveDef_ptr get_primitive(CORBA::PrimitiveKind kind);
```

This method returns a reference to a `PrimitiveKind`.

Parameter	Description
<code>kind</code>	The reference to be returned.

```
CORBA::Contained_ptr lookup_id(const char * search_id);
```

This method searches for an object in the interface repository that matches the specified search id. If no match is found, a `NULL` value is returned.

Parameter	Description
<code>search_id</code>	The identifier to use for the search.

```
CORBA::FixedDef_ptr create_fixed(CORBA::UShort digits, CORBA::Short scale)
```

This method sets the number of digits and the scale for the fixed type.

Parameter	Description
<code>Ushort digits</code>	The number of digits for the fixed type.
<code>short scale</code>	The scale of the fixed type.

SequenceDef

```
class SequenceDef : public CORBA::IDLType, public CORBA::Object
```

The class is used to represent a sequence that is stored in the interface repository. This interface provides methods for setting and retrieving the sequence's bound and element type.

SequenceDef methods

`CORBA::ULong bound()`

This method returns the bounds of the sequence.

`void bound(CORBA::ULong bound)`

This method sets the bound of the sequence.

Parameter	Description
<code>members</code>	The list of members.

`CORBA::TypeCode_ptr element_type();`

This method returns the `TypeCode` of the elements in this sequence.

`CORBA::IDLType_ptr element_type_def();`

This method returns the IDL type of the elements in this sequence.

`void element_type_def(CORBA::IDLType_ptr element_type_def);`

This method sets the IDL type of the elements in this sequence.

Parameter	Description
<code>element_type_def</code>	The IDL type to set elements to.

StringDef

```
class StringDef : public CORBA::IDLType, public CORBA::Object
```

The class is used to describe `Strings` stored in the interface repository. This interface provides methods for setting and retrieving the bounds of the strings.

StringDef methods

`CORBA::ULong bound();`

This method returns the bounds of the `String`.

`void bound(CORBA::ULong bound);`

This method sets the bounds of the `String`.

Parameter	Description
<code>bound</code>	The list of members.

StructDef

```
class StructDef : public CORBA::TypedDef, public CORBA::Container, public CORBA::Object
```

The class is used to represent a structure that is stored in the interface repository.

StructDef methods

```
CORBA::StructMemberSeq *members();
```

This method returns the structure's list of members.

```
void members(CORBA::StructMemberSeq& members);
```

This method sets the structure's list of members.

Parameter	Description
members	The list of members.

StructMember

```
struct CORBA::StructMember
```

This interface is used to define the member for the struct. It uses the name and type variables in the definition.

StructMember methods

```
CORBA::String_var name
```

The name of the type.

```
CORBA::TypeCode_var type
```

The type's IDL type.

```
CORBA::IDLType_var type_def
```

The IDL type's IDL type definition.

TypedefDef

```
class TypedefDef : public CORBA::Contained, public CORBA::IDLType, public
CORBA::Object
```

This abstract base class represents a user-defined structure that is stored in the interface repository. The following interfaces all inherit from this interface:

- AliasDef
- EnumDef
- ExceptionDef
- StructDef
- UnionDef
- WstringDef

TypeDescription

```
structure TypeDescription
```

The `TypeDescription` structure contains the information that describes a type for an operation stored in the interface repository.

TypeDescription members

CORBA::String_var **name**

The name of the type.

CORBA::String_var **id**

The repository id of the type.

CORBA::String_var **defined_in**

The name of the module or interface in which this type is defined.

CORBA::String_var **version**

The type's version.

CORBA::TypeCode_var **type**

The type's IDL type.

UnionDef

```
class UnionDef : public CORBA::TypedefDef, public CORBA::Container, public
CORBA::Object
```

This class is used to represent a `Union` that is stored in the interface repository. This class provides methods for setting and retrieving the union's list of members and discriminator type.

UnionDef methods

```
CORBA::TypeCode_ptr discriminator_type();
```

This method returns the `TypeCode` of the discriminator of the `Union`.

```
CORBA::IDLType_ptr discriminator_type_def();
```

This method returns the IDL type of the `Union`'s discriminator.

```
void discriminator_type_def(CORBA::IDLType_ptr discriminator_type_def);
```

This method sets the IDL type of the `Union`'s discriminator.

Parameter	Description
<code>discriminator_type_def</code>	The list of members.

```
CORBA::UnionMemberSeq *members();
```

This method returns the `Union`'s list of members.

```
void members(CORBA::UnionMemberSeq& members);
```

This method sets the Union's list of members.

Parameter	Description
members	The list of members.

UnionMember

```
struct CORBA::UnionMember
```

The `UnionMember` struct contains information that describes a `Union` that is stored in the interface repository.

UnionMember members

```
CORBA::String_var name
```

The name of the Union.

```
CORBA::Any label
```

The label of the Union.

```
CORBA::TypeCode_var type
```

The Union's typecode.

```
CORBA::IDLType_var type_def
```

The Union's IDL type.

ValueBoxDef

```
class ValueBoxDef public CORBA::Contained, public CORBA::IDLType, public CORBA::Object
```

This interface is used as a simple valuetype that contains a single public member of any IDL type. `ValueBoxDef` is a simplified version of `ValueType`:

```
public valuetype <IDLType> value;
```

This declaration is almost equal to `valuetype boxed type <IDLType>` but `ValueBoxDef` is not the same as simple `ValueTypeDef`.

Methods

```
CORBA::IDLType_ptr original_type_def();
```

This method identifies the type being boxed.

```
void original_type_def(CORBA::IDLType_ptr original_type_def);
```

This method sets the type being boxed.

ValueDef

```
class CORBA::ValueDef public CORBA::Container, public CORBA::Contained, public
CORBA::IDLType, public CORBA::Object
```

This interface describes the IDL value type called a `construct`. This interface is very close to a class type. It represent a value definition that is stored in the Interface Repository.

Methods

```
CORBA::InterfaceDefSeq supported_interfaces( );
```

This method lists the interfaces which this value type supports.

```
void supported_interfaces(const CORBA::interfaceDefSeq& supported_interfaces);
```

This method sets the supported interfaces.

```
CORBA::InitializerSeq& initializers( );
```

This method returns the list of initializers.

```
void initializers(const CORBA::InitializerSeq& initializers);
```

This method sets the initializers.

```
CORBA.ValueDef_ptr base_value( );
```

This method describes the value types from which this value inherits.

```
void base_value(CORBA::ValueDef_ptr base_value);
```

This method sets the value types

```
CORBA.ValueDefSeq& abstract_base_values( );
```

This method returns the list of the abstract value types from which this value inherits.

```
void abstract_base_values(const CORBA::ValueDef[Seq& abstract_base_values);
```

This method defines the abstract value type's base value.

```
CORBA::Boolean is_abstract( );
```

This method returns `true` if the value is an abstract value type.

```
void is_abstract(CORBA::Boolean is_abstract);
```

This method sets the valuetype to be an abstract value type.

```
CORBA::Boolean is_custom( );
```

This method returns `true` if the value uses custom marshalling.

```
void is_custom(CORBA::Boolean is_custom);
```

This method sets the custom marshalling for the value.

```
CORBA::Boolean is_truncatable( ):
```

This method returns `true` if the value can be truncated from its base value safely.

```
void is_truncatable(CORBA::Boolean is_truncatable);
```

This method sets the truncated attribute for this value.

```
CORBA::Boolean is_a(const char* value_id);
```

This method returns `true` if the value on which it is invoked either is identical to or inherits, directly or indirectly from the interface or value defined by the `value_id` parameter. Otherwise it returns `false`.

```
CORBA::ValueDef_ptr FullValueDescription* describe_value();
```

This method returns a `FullValueDescription` describing the value including its operations and attributes.

```
CORBA::ValueMemberDef_ptr create_value_member(const Char* id, const Char* name,  
const Char* version, CORBA::IDLType_ptr type_def, CORBA::short access);
```

This method returns a new `ValueMemberDef` contained in the `ValueDef` on which it is invoked.

Parameter	Description
<code>id</code>	The repository id for this type.
<code>name</code>	The name of this type.
<code>version</code>	The object's version.
<code>type_def</code>	The value's IDL type.
<code>short access</code>	The access value.

```
CORBA::AttributeDef_ptr create_attribute(const Char* id, const Char* name, const  
Char* version, CORBA::IDLType_ptr type, CORBA::AttributeMode mode);
```

This method creates a new attribute definition for this valuetype and returns a new `AttributeDef` for it

Parameter	Description
<code>id</code>	The repository id for this type.
<code>name</code>	The name of this type.
<code>version</code>	The object's version.
<code>type</code>	The type's IDL type.
<code>mode</code>	The object's mode.

```
CORBA::OperationDef_ptr create_operation(const Char* id, const Char* name, const  
Char* version, CORBA::IDLType_ptr result, CORBA::OperationMode mode, const  
CORBA::ParDescriptionSeq& params, const CORBA::ExceptionDefSeq& exceptions, const  
CORBA::ContextIDSeq& contexts);
```

This method creates a new `Operation` for this valuetype and returns an `OperationDef` for it.

Parameter	Description
<code>id</code>	The repository id for this type.
<code>name</code>	The name of this type.
<code>version</code>	The object's version.

Parameter	Description
result	The IDL type of the operation.
mode	The object's mode.
params	The list of the operation's parameters.
exceptions	The list of the operation's exceptions.
contexts	The list of the operation's contexts.

ValueDescription

```
struct CORBA::ValueDescription
```

This interface describes a value type that is stored in the Interface Repository.

Values

```
CORBA::String_var name
```

The name of the type.

```
CORBA::String_var id
```

The repository id of the type.

```
CORBA::Boolean is_abstract
```

If this variable is `true`, the value is an abstract value type.

```
CORBA::Boolean is_custom
```

If this variable is `true`, the valuetype is custom marshalled.

```
CORBA::String_var defined_in.
```

The repository Id of the module in which this type is defined.

```
CORBA::String_var version
```

The type's version.

```
CORBA::RepositoryIdSeq& supported_interfaces
```

The list of interfaces which this value type supports.

```
CORBA::RepositoryIdSeq& abstract_base_values
```

The list of abstract value types from which this value inherits.

```
CORBA::Boolean is_truncatable
```

If this variable is `true`, the value type can be truncated to its base value type safely.

```
CORBA::String_var base_value
```

The value types from which this value inherits.

WstringDef

```
class WstringDef : public CORBA::IDLType, public CORBA::Object
```

This class is used to describe Unicode strings that are stored in the interface repository. It provides methods for setting and retrieving the bounds of a Unicode string.

WStringDef methods

```
CORBA::ULong bound();
```

This method returns the bounds of the `Wstring`.

```
void members(CORBA::ULong bound);
```

This method sets the bounds of the `Wstring`.

Parameter	Description
<code>members</code>	The list of members.

Activation interfaces and classes

This section describes the interfaces and classes used in the activation of object implementations.

ImplementationStatus

```
struct ImplementationStatus
```

ImplementationStatus is used to track the activation state for a server that is registered with the OAD.

```
module Activation
{
    :
    struct ImplementationStatus {
        extension::CreationImplDef impl;
        ObjectStatusList status;
    };
    :
};
```

Include file

Include the `oad_c.hh` file when you use this class.

ImplementationStatus members

```
CreationImplDef impl;
```

The `CreationImplDef` for the object implementation.

```
ObjectStatusList status;
```

Represents a list of status information for each object offered by the server. See [“ObjectStatusList” on page 121](#) for information on the `ObjectStatusList` class.

OAD

The OAD interface provides access to the OAD (Object Activation Daemon). It is used by the administration tools for listing, registering, and unregistering objects. It can also be used by client code for programmatic administration of the OAD.

The following code sample shows the OAD IDL:

```
interface OAD {
    extension::CreationImplDef create_CreationImplDef();

    Object reg_implementation(in extension::CreationImplDef impl)
        raises(DuplicateEntry, InvalidPath);

    extension::CreationImplDef get_implementation(
        in CORBA::RepositoryId repId,
        in COBRA::RepositoryId repId,
        in string object_name)
        raises(NotRegistered);

    void change_implementation(
        in extension::CreationImplDef old_info,
        in extension::CreationImplDef new_info)
        raises(NotRegistered, InvalidPath, IsActive);

    attribute boolean destroy_on_unregister;

    void unreg_implementation(
        in CORBA::RepositoryId repId,
        in string object_name)
        raises(NotRegistered);

    void unreg_interface(in CORBA::RepositoryId repId)
        raises(NotRegistered);

    void unregister_all();

    ImplementationStatus get_status(
        in CORBA::RepositoryId repId,
        in string object_name)
        raises(NotRegistered);

    ImplStatusList get_status_interface(
        in CORBA::RepositoryId repId)
        raises(NotRegistered);

    ImplStatusList get_status_all();

    Object lookup_interface(in CORBA::RepositoryId repId, in long timeout)
        raises(NotRegistered, FailedToExecute,
            NotResponding, Busy);
    Object lookup_implementation(in CORBA::RepositoryId repId, in string
object_name, in long timeout)
        raises(NotRegistered, FailedToExecute,
            NotResponding, Busy);

    extension::CreationImplDef boa_activate_obj(
        in Object obj,
        in string repository_id,
        in long unique_id)
        raises(NotRegistered);
}
```

```

void boa_deactivate_obj(in Object obj,
    in string repository_id,
    in long unique_id)
    raises (NotRegistered);

string generated_command(in extension::CreationImplDef impl);

string generated_environment(in extension::CreationImplDef impl);

};

```

For a complete description of the IDL source codes, refer to the `oad.idl` file located in the VisiBroker installation in the following directory:

```
<install_dir>\idl\
```

Include file

Include the `oad_c.hh` file when you use this class.

OAD methods

```
void change_implementation(const extension::CreationImplDef&_old_info, const
extension::CreationImplDef& _new_info);
```

This method changes an object's implementation dynamically. You can use this method to change the registration's activation policy, path name, argument settings, and environment settings.

Parameter	Description
<code>old_info</code>	The information you want to change.
<code>new_info</code>	The information to replace the old info.

Exception	Description
<code>NotRegistered</code>	The object you specify is not registered. You must specify a registered object.
<code>IsActive</code>	The object implementation is currently running. Deactivate the object and then try to change its information.

Caution You cannot change information for a currently active implementation. Be sure to exercise caution when changing an object's implementation name and object name with this method. Doing so will prevent client applications from locating the object using the old name.

```
CreationImplDef_ptr create_CreationImplDef();
```

Returns an instance of a `extension::CreationImplDef_ptr` object. You can then set its attributes.

```
void destroy_on_unregister(CORBA::Boolean val);
```

Sets the `destroy_on_unregister` attribute for the OAD.

Parameter	Description
<code>val</code>	If set to <code>TRUE</code> , any active implementations are shut down when they are unregistered. Otherwise, they will not be shut down when unregistered.

Note Currently, this attribute cannot be set programatically.

```
CORBA::Boolean destroy_on_unregister();
```

Returns the setting for the `destroy_on_unregister` attribute for an implementation. If the attribute is set to `TRUE`, any active implementations are shut down when unregistered.

Note Currently, this attribute cannot be set programatically.

```
CORBA::CreationImplDef_ptr get_implementation(const char *repId, const char *object_name);
```

This method retrieves information about implementations registered for the specified repository identifier and object name. It returns a `extension::CreationImplDef_ptr` object.

Parameter	Description
<code>repId</code>	The repository identifier.
<code>object_name</code>	The object name.

Exception	Description
<code>NotRegistered</code>	The object you specify is not registered. You must specify a registered object.

```
ImplementationStatus *get_status(const char *repId, const char *object_name);
```

This method retrieves the status information about implementations registered for the specified repository identifier and object name.

Parameter	Description
<code>repId</code>	The repository identifier.
<code>object_name</code>	The object name.

```
ImplStatusList *get_status_all();
```

Returns an `ImplStatusList` containing the status information for all implementations.

```
ImplStatusList *get_status_interface(const char *repId);
```

This method gets the status information about implementations registered for the specified repository identifier.

Parameter	Description
<code>repId</code>	The repository identifier.

Exception	Description
<code>NotRegistered</code>	The object you specify is not registered. You must specify a registered object.

```
::CORBA::Object_ptr reg_implementation(const extension::CreationImplDef& _impl);
```

This method registers an implementation with the OAD and the VisiBroker Edition directory service.

Parameter	Description
<code>impl</code>	The instance of <code>CreationImplDef</code> .

Exception	Description
<code>DuplicateEntry</code>	The object you specify is a duplicate entry. You must specify an unregistered object.

```
void unreg_implementation(const char *repId, const char *object_name);
```

This method unregisters implementations by repository identifier and object name. If the `destroy_on_unregister` attribute is set to `true`, this method terminates all processes currently implementing the repository identifier and object name that is specified.

Parameter	Description
<code>repId</code>	The repository identifier.
<code>object_name</code>	The object name.

Exception	Description
<code>NotRegistered</code>	The object you specify is not registered. You must specify a registered object.

```
void unreg_interface(const char *repId);
```

This method unregisters all implementations for a repository identifier. If the `destroy_on_unregister` attribute is set to `true`, this method terminates all processes currently implementing the repository identifier specified.

Parameter	Description
<code>repId</code>	The repository identifier.

Exception	Description
<code>NotRegistered</code>	The object you specify is not registered. You must specify a registered object.

```
void unregister_all();
```

This method unregisters all implementations. Unless the attribute `destroy_on_unregister` is set to `true`, all active implementations continue to execute.

ObjectStatus

```
struct ObjectStatus
```

This structure is used to store information about a particular object offered by an object implementation that is registered with the OAD. This structure is returned by the `ObjectStatusList` class, described in [“ObjectStatusList” on page 121](#).

```
    module Activation
    {
        :
        struct ObjectStatus {
            long        unique_id;
            State       activation_state;
            Object      objRef;
        };
        :
    };
```

Include file

Include the `oad_c.hh` file when you use this class.

ObjectStatus members

`CORBA::Long unique_id;`

A unique identifier for the object.

`State activation_state;`

The object's current activation state; one of these values:

- `ACTIVE`
- `INACTIVE`
- `WAITING_FOR_ACTIVATION`

`CORBA::Object objRef;`

The object whose state is represented in the structure.

ObjectStatusList

class `ObjectStatusList`

This class implements a list of `ObjectStatus` structures and is used to represent information about the objects offered by a server.

See also

- [“ObjectStatus” on page 120](#)

Include file

Include the `oad_c.hh` file when you use this class.

ObjectStatusList methods

void `length`(CORBA::ULong `len`);

Sets the length of the list.

Parameter	Description
<code>len</code>	The length of the list.

CORBA::ULong `length`() const;

Returns the length of the list.

CORBA::ULong `maximum`() const;

Returns the maximum length of the list.

ObjectStatus& `operator[]`(CORBA::ULong `index`);

Returns the `ObjectStatus` structure with the specified index in the list.

Parameter	Description
<code>index</code>	The zero-based index of the item in the list.

Naming Service (VisiNaming) interfaces and classes

This section describes the interfaces and classes for the VisiBroker Naming Service (VisiNaming).

NamingContext

```
class _VISNMEEXPORT NamingContext : public virtual CORBA_Object
```

This object is used to contain and manipulate a list of names that are bound to the VisiBroker ORB objects or to other `NamingContext` objects. Client applications use this interface to resolve or list all of the names within that context. Object implementations use this object to bind names to object implementations or to bind a name to a `NamingContext` object. The code sample below shows the IDL specification for the `NamingContext`.

```
module CosNaming {
    interface NamingContext {
        void bind(in Name n, in Object obj)
            raises(NotFound, CannotProceed, InvalidName, AlreadyBound);
        void rebind(in Name n, in Object obj)
            raises(NotFound, CannotProceed, InvalidName);
        void bind_context(in Name n, in NamingContext nc)
            raises(NotFound, CannotProceed, InvalidName, AlreadyBound);
        void rebind_context(in Name n, in NamingContext nc)
            raises(NotFound, CannotProceed, InvalidName);
        Object resolve(in Name n)
            raises(NotFound, CannotProceed, InvalidName);
        void unbind(in Name n)
            raises(NotFound, CannotProceed, InvalidName);
        NamingContext new_context();
        NamingContext bind_new_context(in Name n)
            raises(NotFound, CannotProceed, InvalidName, AlreadyBound);
        void destroy()
            raises(NotEmpty);
        void list(in unsigned long how_many,
                out BindingList bl,
                out BindingIterator bi);
    };
};
```

NamingContext methods

```
virtual void bind(const Name& -n, CORBA::Object _ptr _obj): raises(NotFound,
CannotProceed, InvalidName, AlreadyBound);
```

This method attempts to bind the specified `Object` to the specified `Name` by resolving the context associated with the first `NameComponent` and then binding the object to the new context using the following `Name`:

```
Name [NameComponent<sub>2</sub>, ... ,NameComponent<sub>(n-1)</sub> /
sub>,NameComponent<sub>n</sub>]
```

This recursive process of resolving and binding continues until the context associated with the `NameComponent` ($n-1$) is resolved and the actual name-to-object binding is stored. If parameter `n` is a simple name, the `obj` will be bound to `n` within this `NamingContext`.

Parameter	Description
<code>n</code>	A <code>Name</code> , initialized with the desired name for the object.
<code>obj</code>	The object to be named.

This method may raise the following exceptions:

Exception	Description
<code>NotFound</code>	The <code>Name</code> , or one of its components, could not be found.
<code>CannotProceed</code>	One of the <code>NameComponent</code> objects in the sequence could not be resolved. The client may still be able to continue the operation from the returned naming context.
<code>InvalidName</code>	The specified <code>Name</code> has no name components or the <code>id</code> field of one of its name components is an empty string.
<code>AlreadyBound</code>	The <code>Name</code> on a <code>bind</code> or <code>bind_context</code> operation has already been bound to another object within the <code>NamingContext</code> .

```
virtual void rebind(const Name& _n, CORBA::Object _ptr _obj) raises(NotFound,
CannotProceed, InvalidName);
```

This method is exactly the same as the `bind` method, except that it never raises the `AlreadyBound` exception. If the specified `Name` has already been bound to another object, this method replaces that binding with the new binding.

Parameter	Description
<code>n</code>	A <code>Name</code> structure, initialized with the desired name for the object.
<code>obj</code>	The object to be named.

The following exceptions may be raised by this method.

Exception	Description
<code>NotFound</code>	The <code>Name</code> , or one of its components, could not be found.
<code>CannotProceed</code>	One of the <code>NameComponent</code> objects in the sequence could not be resolved. The client may still be able to continue the operation from the returned naming context.
<code>InvalidName</code>	The specified <code>Name</code> has no name components or the <code>id</code> field of one of its name components is an empty string.

```
virtual void bind_context(const Name& _n, NamingContext_ptr _nc) raises(NotFound,
CannotProceed, InvalidName, AlreadyBound);
```

This method is identical to the `bind` method, except that it associates the supplied Name with a `NamingContext`, not an arbitrary `VisiBroker ORB` object.

Parameter	Description
<code>n</code>	A <code>Name</code> structure initialized with the desired name for the naming context. The first $(n-1)$ <code>NameComponent</code> structures in the sequence must resolve to a <code>NamingContext</code> .
<code>nc</code>	The <code>NamingContext</code> object to be bound.

The following exceptions may be raised by this method.

Exception	Description
<code>NotFound</code>	The <code>Name</code> , or one of its components, could not be found.
<code>CannotProceed</code>	One of the <code>NameComponent</code> objects in the sequence could not be resolved. The client may still be able to continue the operation from the returned naming context.
<code>InvalidName</code>	The specified <code>Name</code> has no name components or the <code>id</code> field of one of its name components is an empty string.
<code>AlreadyBound</code>	The <code>Name</code> on a <code>bind</code> or <code>bind_context</code> operation has already been bound to another object within the <code>NamingContext</code> .

```
virtual void rebind_context(const Name& _n, NamingContext_ptr _nc)
raises(NotFound, CannotProceed, InvalidName);
```

This method is exactly the same as the `bind_context` method, except that this method never raises the `AlreadyBound` exception. If the specified `Name` has already been bound to another naming context, this method replaces that binding with the new binding.

Parameter	Description
<code>n</code>	A <code>Name</code> structure, initialized with the desired name for the object.
<code>nc</code>	The <code>NamingContext</code> object to be rebound.

The following exceptions may be raised by this method.

Exception	Description
<code>NotFound</code>	The <code>Name</code> , or one of its components, could not be found.
<code>CannotProceed</code>	One of the <code>NameComponent</code> objects in the sequence could not be resolved. The client may still be able to continue the operation from the returned naming context.
<code>InvalidName</code>	The specified <code>Name</code> has no name components or the <code>id</code> field of one of its name components is an empty string.

```
virtual CORBA::Object_ptr resolve(const Name& _n) raises(NotFound, CannotProceed,
InvalidName);
```

This method attempts to resolve the specified `Name` and return an object reference. If parameter `n` is a *simple name*, it is resolved relative to this `NamingContext`.

If `n` is a *complex name*, it is resolved using the context associated with the first `NameComponent`. Next, the new context to resolve the following `Name`:

```
Name [NameComponent<sub>(2)</sub>, ..., NameComponent<sub>(n-1)</sub>,
NameComponent<sub>n</sub>]
```

This recursive process continues until the object associated with the *n*th NameComponent is returned.

Parameter	Description
n	A Name structure, initialized with the name for the desired object.

The following exceptions may be raised by this method.

Exception	Description
NotFound	The Name, or one of its components, could not be found.
CannotProceed	One of the NameComponent objects in the sequence could not be resolved. The client may still be able to continue the operation from the returned naming context.
InvalidName	The specified Name has no name components or the id field of one of its name components is an empty string.

```
virtual void unbind(const Name& _n) raises(NotFound, CannotProceed, InvalidName);
```

This method performs the inverse operation of the bind method, removing the binding associated with the specified Name.

Parameter	Description
n	A Name structure, initialized with the desired name to be unbound.

The following exceptions may be raised by this method.

Exception	Description
NotFound	The Name, or one of its components, could not be found.
CannotProceed	One of the NameComponent objects in the sequence could not be resolved. The client may still be able to continue the operation from the returned naming context.
InvalidName	The specified Name has no name components or the id field of one of its name components is an empty string.

```
virtual NamingContext_ptr new_context();
```

This method creates a new naming context. The newly created context is implemented within the same server as this object. The new context is initially not bound to any Name.

```
virtual NamingContext_ptr bind_new_context(const Name& _n) raises(NotFound,
CannotProceed, InvalidName, AlreadyBound);
```

This method creates a new context and binds it to the specified `Name` within this `Context`.

Parameter	Description
<code>n</code>	A <code>Name</code> structure, initialized with the specified <code>Name</code> for the newly created <code>NamingContext</code> object.

The following exceptions can be raised by this method.

Exception	Description
<code>NotFound</code>	The <code>Name</code> or one of its components could not be found.
<code>CannotProceed</code>	One of the <code>NameComponent</code> objects in the sequence could not be resolved. The client may still be able to continue the operation from the returned <code>NamingContext</code> .
<code>InvalidName</code>	The specified <code>Name</code> has no name components or the <code>id</code> field of one of its name components is an empty string.
<code>AlreadyBound</code>	The <code>Name</code> on a <code>bind</code> or <code>bind_context</code> operation has already been bound to another object within the <code>NamingContext</code> .

```
virtual void destroy() raises(NotEmpty);
```

This method deactivates this naming context. Any subsequent attempt to invoke operations on this object raises a `CORBA::OBJECT_NOT_EXIST` runtime exception.

Before using this method, all `Name` objects that have been bound relative to this `NamingContext` should be unbound using the `unbind` method. Any attempt to destroy a `NamingContext` that is not empty raises a `NotEmpty` exception.

```
virtual void list(CORBA::ULong _how_many, BindingList_out _bl, BindingIterator_out
_bi)
```

This method returns all of the bindings contained by this context. A maximum of `how_many` `Names` are returned with the `BindingList`. Any left over bindings are returned via the `BindingIterator`. The returned `BindingList` and `BindingIterator` are described in detail in [“Binding and BindingList” on page 129](#) and can be used to navigate the list of names.

Parameter	Description
<code>how_many</code>	The maximum number of <code>Names</code> to be returned.
<code>bl</code>	A list of <code>Names</code> returned to the caller. The number of names in the list will not exceed <code>how_many</code> .
<code>bi</code>	An iterator for use in traversing the rest of the <code>Names</code> .

NamingContextExt

```
class _VISNMEEXPORT NamingContextExt : public virtual NamingContext, public virtual
CORBA Object
```

The NamingContextExt interface, which extends NamingContext, provides the operations required to use stringified names and URLs.

This code sample shows the IDL Specification for the NamingContextExt interface.

```
module CosNaming {
    interface NamingContextExt {
        typedef string StringName;
        typedef string Address;
        typedef string URLString;
        StringName to_string(in Name n)
            raises(InvalidName);
        Name to_name(in StringName sn)
            raises(InvalidName);
        exception InvalidAddress {};
        URLString to_url(in Address addr, in StringName sn)
            raises(InvalidAddress, InvalidName);
        Object resolve_str(in StringName n)
            raises(NotFound, CannotProceed, InvalidName, AlreadyBound);
    };
};
```

NamingContextExt methods

```
virtual char* to_string(const Name& _n) raises(InvalidName);
```

This operation returns the stringified representation of the specified Name.

Parameter	Description
n	A Name structure initialized with the desired name for object.

The following exceptions can be raised by this method.

Exception	Description
InvalidName	The specified Name has no name components or the id field of one of its name components is an empty string.

```
virtual Name* to_name(const char* _sn); raises(InvalidName);
```

This operation returns a Name object for the specified stringified name.

Parameter	Description
_sn	The stringified name of an object.

The following exceptions can be raised by this method.

Exception	Description
InvalidName	The specified Name has no name components or the id field of one of its name components is an empty string.


```
virtual char* to_url(const char* _addr, const char* _sn); raises(InvalidAddress,
InvalidName);
```

This operation returns a fully-formed string URL using the URL specified in `_addr` and the stringified name in `_sn`.

Parameter	Description
<code>_addr</code>	A URL component of the form <code>myhost.borland.com:800</code> . If the Address is empty, it is the local host.
<code>_sn</code>	A stringified name of an object.

The following exceptions can be raised by this method.

Exception	Description
<code>InvalidAddress</code>	The specified Address is malformed.
<code>InvalidName</code>	The specified Name has no name components or the <code>id</code> field of one of its name components is an empty string.

```
virtual CORBA::Object _ptr resolve_str(const char* _n);
raises(NotFound, CannotProceed, InvalidName, AlreadyBound);
```

This operation returns a `Name` object for the specified stringified name.

Parameter	Description
<code>_n</code>	A stringified name of an object.

The following exceptions can be raised by this method.

Exception	Description
<code>NotFound</code>	The Name, or one of its components, could not be found.
<code>CannotProceed</code>	One of the <code>NameComponent</code> objects in the sequence could not be resolved. The client may still be able to continue the operation from the returned <code>NamingContext</code> .
<code>InvalidName</code>	The specified Name has no name components or the <code>id</code> field of one of its name components is an empty string.
<code>AlreadyBound</code>	The Name on a <code>bind</code> or <code>bind_context</code> operation has already been bound to another object within the <code>NamingContext</code> .

Binding and BindingList

The `Binding`, `BindingList`, and `BindingIterator` interfaces are used to describe the name-object bindings contained in a `NamingContext`. The `Binding` struct encapsulates one such pair. The `binding_name` field represents the `Name` and the `binding_type` indicates whether the `Name` is bound to a `VisiBroker` ORB object or a `NamingContext` object.

The `BindingList` is a sequence of `Binding` structures contained by a `NamingContext` object. An example program that uses the `BindingList` can be found in the `Naming Service` section of the *Borland VisiBroker Developer's Guide*.

This code sample shows the IDL specification for the Binding structure.

```
module CosNaming {
    enum BindingType {
        nobject,
        ncontext
    }
    struct Binding {
        Name binding_name;
        BindingType binding_type;
    };
    typedef sequence<Binding> BindingList;
};
```

BindingIterator

```
class _VISNMEEXPORT BindingIterator : public virtual CORBA_Object
```

This object allows a client application to walk through the unbounded collection of bindings returned by the NamingContext operation list. An example program that uses the BindingIterator can be found in Naming Service section of the Borland VisiBroker *Developer's Guide*.

This code sample shows the IDL specification for the BindingIterator interface.

```
module CosNaming {
    interface BindingIterator {
        boolean next_one(out Binding b);
        boolean next_n(in unsigned long how_many, out BindingList b);
        void destroy();
    };
};
```

BindingIterator methods

```
virtual CORBA::Boolean next_one(Binding_out b_);
```

This method returns the next Binding from the collection. It returns CORBA::FALSE if the list has been exhausted. Otherwise, it returns CORBA::TRUE.

Parameter	Description
b_	The next Binding object from the list.

```
virtual CORBA::Boolean next_n(CORBA::ULong _how_many, BindingList_out _b);
```

This method returns a BindingList containing the number of requested Binding objects from the list. The number of bindings returned may be less than the requested amount if the list is exhausted. CORBA::FALSE is returned when the list has been exhausted. Otherwise, CORBA::TRUE is returned.

Parameter	Description
_how_many	The maximum number of Binding object desired.
_b	A BindList containing no more than the requested number of Binding objects.

```
virtual void destroy();
```

This method destroys this object and releases the memory associated with the object. Failure to call this method will result in increased memory usage.

NamingContextFactory

```
class _VISNMEEXPORT NamingContextFactory : public virtual CORBA_Object
```

This interface is provided to instantiate an initial NamingContext. A client may bind to an object of this type and use the `create_context` method to create an initial context. Once the initial context has been created, the `new_context` method can be used to create other contexts. An instance of this naming context factory is created when the naming service is started, described in the Naming Service section of the Borland Enterprise Server *Developer's Guide*.

To create an initial NamingContextFactory that automatically creates a single root context, see [“ExtendedNamingContextFactory” on page 132](#).

This code sample shows the IDL specification for the NamingContextFactory.

```
module CosNaming {
    interface NamingContextFactory {
        NamingContext create_context();
        oneway void shutdown();
    };
};
```

Methods

```
virtual CosNaming::NamingContextEXT_ptr create_context();
```

This method allows a client to create a naming context. Since the specification for naming contexts states that they do not have any notion of a root context, simply instantiating a NamingContextFactory does not create a naming context.

```
virtual ClusterManager_ptr get_cluster_manager();
```

This method returns the cluster.

```
virtual NamingContextList* list_all_roots (const char* _password);
```

This method returns the list of all root contexts.

```
virtual void remove_stale_contexts (const char* _password);
```

This method allows the client to remove members from the cluster during the cluster's lifetime.

```
virtual void shutdown();
```

This method allows a client to gracefully shut the naming service down. If the service is restarted with the same logfile, the factory is restored to the state it had prior to being shut down.

ExtendedNamingContextFactory

```
class _VISNMEEXPORT ExtendedNamingContextFactory : public virtual  
NamingContextFactory, public virtual CORBA_Object
```

This interface extends the `NamingContextFactory` interface and allows the creation of a default root within a factory when the extended naming service is started, described in the Naming Service section of the Borland Enterprise Server *Developer's Guide*.

This code sample shows the IDL specification for the `ExtendedNamingContextFactory`.

```
module CosNaming {  
    interface ExtendedNamingContextFactory : NamingContextFactory{  
        NamingContext root_context();  
    };  
};
```

Methods

```
virtual CosNaming::NamingContextExt_ptr root_context();
```

This method returns the root naming context that was created automatically when this object was instantiated.

Event service interfaces and classes

This section describes the interfaces and classes for the VisiBroker for C++ Event Service.

ConsumerAdmin

```
public interface ConsumerAdmin extends ConsumerAdminPOA
```

This interface is used by consumer applications to obtain a reference to a proxy supplier object. This is the second step in connecting a consumer application to an EventChannel.

IDL definition

```
module CosEventChannelAdmin {  
    interface ConsumerAdmin {  
        ProxyPushConsumer obtain_push_supplier();  
        ProxyPullConsumer obtain_pull_supplier();  
    };  
};
```

ConsumerAdmin methods

```
public ProxyPushSupplier obtain_push_supplier();
```

The `obtain_push_supplier` method should be invoked if the calling consumer application is implemented using the push model. If the application is implemented using the pull model, the `obtain_pull_supplier` method should be invoked.

```
public ProxyPullSupplier obtain_pull_supplier();
```

The returned reference is used to invoke either the `connect_push_consumer`, described in “[ProxyPushConsumer](#)” on page 136, or the `connect_pull_consumer` method, described in “[ProxyPullConsumer](#)” on page 136.

EventChannel

public interface **EventChannel**

The `EventChannel` provides the administrative operations for adding suppliers and consumers to the channel and for destroying the channel. For information on creating an event channel, see “[EventChannelFactory](#)” on page 135.

Suppliers and consumers both use the `_bind` method to obtain an `EventChannel` reference. As with any `_bind` invocation, the caller can optionally specify the object name of the desired `EventChannel` as well as any desired bind options. These arguments can be passed to the supplier or consumer as initial parameters or they may be obtained from the Naming Service, if it is available. If the object name is not specified, `VisiBroker` locates a suitable `EventChannel`. Once a supplier or consumer is connected to an `EventChannel`, it may invoke any of the `EventChannel` methods.

Methods

The following code sample shows the supplier binding to an `EventChannel` with the object name “power”.

```
int main(int argc, char* const* argv)
{
    :
    CosEventChannelAdmin::EventChannel_var my_channel =
        CosEventChannelAdmin::EventChannel::_bind("power");
    CosEventChannelAdmin::SupplierAdmin_var = channel->for_suppliers();
    :
}
```

`ConsumerAdmin for_consumers();`

This method returns a `ConsumerAdmin` object that can be used to add consumers to this `EventChannel`.

`SupplierAdmin for_suppliers();`

This method returns a `SupplierAdmin` object that can be used to add suppliers to this `EventChannel`.

`void destroy();`

This method destroys this `EventChannel`.

EventChannelFactory

public interface **EventChannelFactory**

The `EventChannelFactory` provides methods for creating, locating, and destroying event channels.

IDL definition

```
module CosEventChannelAdmin {
    interface EventChannelFactory {
        exception AlreadyExists();
        exception ChannelsExist();
        EventChannel create();
        EventChannel create_by_name(in string name)
            raises(AlreadyExists);
        EventChannel lookup_by_name(in string name);
        void destroy()
            raises(ChannelsExist);
    };
};
```

EventChannelFactory methods

EventChannel **create**();

This method creates an anonymous, transient event channel.

EventChannel **create_by_name**(in string name) raises(AlreadyExists);

This method creates a named, persistent event channel. If an event channel with the specified name has already been created, an `AlreadyExists` exception is raised.

EventChannel **lookup_by_name**(in string name);

This method attempts to return the `EventChannel` with the specified name. If no channel with the specified name exists, a `NULL` value is returned.

void **destroy**();

This method destroys this event channel. The disconnect methods of all suppliers and consumers connected to the channel are called before the channel is destroyed. Once destroyed, if the channel was created by the `create_by_name` method, it is no longer found by the `lookup_by_name` method.

ProxyPullConsumer

```
public interface ProxyPullConsumer
```

This interface is used by a pull supplier application and provides the `connect_pull_supplier` method for connecting the supplier's `PullSupplier`-derived object to the `EventChannel`. An `AlreadyConnected` exception is raised if an attempt is made to connect the same proxy more than once.

IDL definition

```
module CosEventChannelAdmin {
    exception AlreadyConnected();
    interface ProxyPullConsumer : CosEventComm::PullConsumer {
        void connect_pull_supplier(in CosEventComm::PullSupplier pull_supplier)
            raises(AlreadyConnected);
    };
};
```

ProxyPushConsumer

```
public interface ProxyPushConsumer
```

This interface is used by a push supplier application and provides the `connect_push_supplier` method which is used to connect the supplier's `PushSupplier`-derived object to the `EventChannel`. An `AlreadyConnected` exception is raised if an attempt is made to connect the same proxy more than once.

IDL definition

```
module CosEventChannelAdmin {
    exception AlreadyConnected();
    interface ProxyPushConsumer : CosEventComm::PushConsumer {
        void connect_push_supplier(in CosEventComm::PushSupplier push_supplier)
            raises(AlreadyConnected);
    };
};
```

ProxyPullSupplier

```
public interface ProxyPullSupplier
```

This interface is used by a pull consumer application and provides the `connect_pull_consumer` method which is used for connecting the consumer's `PullConsumer`-derived object to the `EventChannel`. An `AlreadyConnected` exception is raised if an attempt is made to connect the same `PullConsumer` more than once.

IDL definition

```

module CosEventChannelAdmin {
    exception AlreadyConnected();
    interface ProxyPullSupplier : CosEventComm::PullSupplier {
        void connect_pull_consumer(in CosEventComm::PullConsumer pull_consumer)
            raises(AlreadyConnected);
    };
};

```

ProxyPushSupplier

public interface **ProxyPushSupplier**

This interface is used by a push consumer application and provides the `connect_push_consumer` method which is used to connect the consumer's PushConsumer-derived object to the EventChannel. An `AlreadyConnected` exception is raised if an attempt is made to connect the same PushConsumer more than once.

IDL definition

```

module CosEventChannelAdmin {
    exception AlreadyConnected();
    interface ProxyPushSupplier : CosEventComm::PushSupplier {
        void connect_push_consumer(in CosEventComm::PushConsumer push_consumer)
            raises(AlreadyConnected);
    };
};

```

PullConsumer

public interface **PullConsumer**

This interface is used to derive consumer objects that use the pull model of communication. The `pull` method is called by a consumer whenever it wants data from the supplier. A `Disconnected` exception is raised if the supplier has disconnected.

The `disconnect_push_consumer` method is used to deactivate this consumer if the channel is destroyed.

IDL definition

```

module CosEventChannelAdmin {
    exception Disconnected {};
    interface PushConsumer {
        void push(in any data) raises(Disconnected);
        void disconnect_push_consumer();
    };
};

```

PushConsumer

```
public interface PushConsumer
```

This interface is used to derive consumer objects that use the push model of communication. The `push` method is used by a supplier whenever it has data for the consumer. A `Disconnected` exception is raised if the consumer has disconnected.

IDL definition

```
module CosEventComm {
    exception Disconnected();
    interface PushConsumer {
        void push(in any data) raises(Disconnected);
        void disconnect_push_consumer();
    };
};
```

PullSupplier

```
public interface PullSupplier
```

This interface is used to derive supplier objects that use the pull model of communication.

IDL definition

```
module CosEventComm {
    interface PullSupplier {
        any pull() raises(Disconnected);
        any try_pull(out boolean has_event) raises(Disconnected);
        void disconnect_pull_supplier();
    };
};
```

PullSupplier methods

```
any pull();
```

This method blocks until there is data available from the supplier. The data is returned an `Any` type. If the consumer has disconnected, this method raises a `Disconnected` exception.

```
any try_pull(out boolean has_event);
```

This non-blocking method attempts to retrieve data from the supplier. When this method returns, `has_event` is set to the value `true` and the data is returned as an `Any` type if there was data available. If the value of `has_event` is `false`, then no data is available and the return value is `NULL`.

```
void disconnect_pull_supplier();
```

This method deactivates this pull server if the channel is destroyed.

PushSupplier

```
public interface PushSupplier
```

This interface is used to derive supplier objects that use the push model of communication. The `disconnect_push_supplier` method is used by the `EventChannel` to disconnect supplier when it is destroyed.

IDL definition

```
module CosEventComm {
    exception AlreadyConnected();
    interface PushSupplier {
        void disconnect_push_supplier();
    };
};
```

SupplierAdmin

```
public interface SupplierAdmin
```

This interface is used by supplier applications to obtain a reference to the proxy consumer object. This is the second step in connecting a supplier application to an `EventChannel`.

IDL definition

```
module CosEventChannelAdmin {
    interface SupplierAdmin {
        ProxyPushConsumer obtain_push_consumer();
        ProxyPullConsumer obtain_pull_consumer();
    };
};
```

```
public ProxyPushConsumer obtain_push_consumer();
```

The `obtain_push_consumer` method should be invoked if the supplier application is implemented using the push model. If the application is implemented using the pull model, the `obtain_pull_consumer` method should be invoked.

```
public ProxyPullConsumer obtain_pull_consumer();
```

The returned reference is used to invoke either the `connect_push_supplier`.

Server Manager Interfaces and Classes

This section describes the VisiBroker for C++ Server Manager interfaces and classes. For additional information about the Server Manager, see “Using the VisiBroker Server Manager” in the *VisiBroker for C++ Developer's Guide*.

The Container Interface

A container can hold properties, operations, and other containers. Each major ORB component is represented as a container. The top-level container corresponds to the ORB itself and includes a few ORB properties, the `shutdown` method, and a few other commonly used containers like `RootPOA` and `Agent`.

The Container Interface

This section explains the C++ methods that can be executed on the container interface. There are four categories:

- Methods related to property manipulation and queries
- Methods related to operations
- Methods related to children containers
- Methods related to storage

Methods related to property manipulation and queries

```
virtual CORBA::StringSequence* list_all_properties();
```

Returns the names of all the properties in the container as a `StringSequence`.

```
virtual PropertySequence* get_all_properties();
```

Returns the `PropertySequence` containing the names, values, and read-write status of all the properties in the container.

```
virtual Property* get_property(const char * name);
```

Returns the value of the property *name* passed as an input parameter.

Parameter	Description
name	The name of the property.

It throws *NameInvalid exception* if the parameter passed is not a valid property name.

```
virtual void set_property(const char* name, CORBA::Any& value);
```

Sets the value of the property *name* to the requested *value*.

Parameters	Descriptions
name	The name of the property whose value is to be set.
value	The property value as <i>Any</i> type.

It throws *NameInvalid*, *ValueInvalid* or *ValueNotSettable* exception.

```
virtual void persist_properties(CORBA::Boolean recurse);
```

Causes the container to actually store its properties to the associated storage. If no storage is associated with the container, a *StorageException* will be raised. When it is invoked with the parameter *recurse=true*, the properties of the children containers are also stored into the storage. It is up to the container to decide if it has to store all the properties or only the changed properties.

Parameter	Description
recurse	Indicates whether the sub-containers' <i>persist_properties</i> should be called recursively.

It can throw *StorageException* exception.

```
virtual void restore_properties(CORBA::Boolean recurse);
```

Instructs the container to obtain its properties from the storage. A container knows exactly what properties it manages and it attempts to read those properties from the storage. The containers shipped with the ORB do not support restoring from the storage. You must create containers that support this feature yourself.

Parameter	Description
recurse	Indicates whether the sub-containers' <i>restore_properties</i> should be called recursively.

It can throw *StorageException* exception.

Methods related to operations

```
virtual CORBA::StringSequence* list_all_operations();
```

Returns the names of all the operations supported in the container.

```
virtual OperationSequence* get_all_operations();
```

Returns all the operations along with the parameters and the type code of the parameters so that the operation can be invoked with the appropriate parameters.

```
virtual Operation* get_operation(const char* name);
```

Returns the parameter information of the operation specified by *name* which can be used to invoke the operation.

Parameter	Description
name	The name of the operation to get the parameter information.

It can throw *NameInvalid* exception if the parameter specifies an operation which is not supported.

```
CORBA::Any* do_operation(const Operation& op);
```

Invokes the method in the operation and returns the result.

Parameter	Description
op	The operation which is to be performed on the server.

It can throw *NameInvalid*, *ValueInvalid* or *OperationFailed*.

Methods related to children containers

```
virtual CORBA::StringSequence* list_all_containers();
```

Returns the names of all the children containers of the current container.

```
virtual NamedContainerSequence* get_all_containers();
```

Returns all the children containers.

```
virtual NamedContainer* get_container(const char * name) ;
```

Returns the child container identified by the *name* parameter.

Parameter	Description
name	The name of the container on which the children containers are to be queried.

If there is no child container with this name, a *NameInvalid* exception is raised.

```
virtual void add_container(const NamedContainer& container);
```

Adds the container as a child container of this *container*.

Parameter	Description
container	The name of the child container to be added into this container.

It can throw *NameAlreadyPresent* or *ValueInvalid* exceptions.

```
virtual void set_container (const char * name, Container_ptr value);
```

Modifies the child container identified by the *name* parameter to one in the *value* parameter.

Parameter	Description
name	The name of the container whose value is to be replaced.
value	The new child container.

It can throw *NameInvalid*, *ValueInvalid* or *ValueNotSettable* exceptions.

Methods related to storage

```
virtual void set_storage(Storage_ptr s, CORBA::Boolean recurse);
```

Sets the storage of this container. If `recurse=true`, it also sets the storage for all its children as well.

Parameter	Description
<code>s</code>	The new storage to be set.
<code>recurse</code>	Indicates whether to set the storage recursively for the children containers.

```
virtual Storage_ptr get_storage();
```

Returns the current storage of the container.

The Storage Interface

The Server Manager provides an abstract notion of *storage* that can be implemented in any fashion. Individual containers may choose to store their properties in databases, flat files, or some other means. The storage implementation included with the VisiBroker ORB uses a flat-file-based approach.

Storage Interface Methods for C++

```
virtual void open();
```

Opens the storage and makes it ready for reading and writing the properties. For the database-based implementation, logging into the database is performed in this method.

It can throw *StorageException* if the storage could not be opened for any reasons.

```
virtual void close();
```

Closes the storage. This method also updates the storage with any properties that have been changed since the last `Container::persist_properties` call. In database implementations, this method closes the database connection.

It can throw *StorageException* if the closing fails for any reasons.

```
virtual Container::PropertySequence* read_properties();
```

Reads all the properties from the storage. It can throw *StorageException* if the properties could not be read from the Storage.

```
virtual Container::Property* read_property(const char * propertyName);
```

Returns the property value for *propertyName* read from the storage.

Parameter	Description
<code>propertyName</code>	The name of the property which is to be read from the Storage.

It can throw *StorageException* or *Container::NameInvalid*.


```
virtual void write_properties( const Container::PropertySequence& p);
```

Saves the property sequence into the storage.

Parameter	Description
p	The sequence of properties which have been changed in the session.

It can throw *StorageException*.

```
virtual void write_property( const Container::Property& p);
```

Saves the single property into the storage.

Parameter	Description
p	The property which is to be written to the persistent storage.

It can throw *StorageException*.

Chapter 10

Transaction Service interfaces and classes

This section describes the following VisiBroker VisiTransact Transaction Service modules, interfaces, and classes:

- CosTransactions and VISTransactions modules
- Current interface
- TransactionalObject interface
- TransactionFactory interface
- Control interface
- Terminator interface
- Coordinator interface
- RecoveryCoordinator interface
- Resource interface
- Synchronization interface
- VISTransactionService class
- VISSessionManager module
- ConnectionPool interface
- Connection interface
- The ITSDDataConnection class
- Native handle acquisition interface
- Local transaction connection and completion interface
- Global transaction connection and completion interface

CosTransactions and VISTransactions modules

This section introduces the `CosTransactions` and `VisTransactions` modules, and describes the data types, structures, and exceptions for the `CosTransactions` module.

Looking at the CosTransactions module

The `CosTransactions` module is the Transaction Service IDL that conforms to the final OMG Transaction Service document. This is the module to use to restrict yourself strictly to CORBA-compliant methods. The IDL for this module is contained in the file **`CosTransactions.idl`**.

You might also consider using the `VISTransactions` module, which contains the IDL for some VisiBroker VisiTransact extensions to the standard. The IDL for the `VISTransactions` module is contained in the file **`VISTransactions.idl`**. You can use **`VISTransactions.idl`** in your code to obtain both the `CosTransactions` and `VISTransactions` modules. For more information, see [“Looking at the VISTransactions module” on page 151](#).

Data types

The `CosTransactions` module defines the data types `enum Status` and `enum Vote`.

The definition for the `enum Status` data type is:

```
enum Status
{
    StatusActive,
    StatusMarkedRollback,
    StatusPrepared,
    StatusCommitted,
    StatusRolledBack,
    StatusUnknown,
    StatusNoTransaction,
    StatusPreparing,
    StatusCommitting,
    StatusRollingBack,
};
```

For a description of each `Status` value, see [“Status value definitions” on page 157](#).

The `enum Vote` data type is used only by implementations of the `CosTransactions::Resource` interface. It is used to indicate the result of a Resource's attempt to prepare a transaction.

The definition for the `enum Vote` data type is:

```
enum Vote
{
    VoteCommit,
    VoteRollback,
    VoteReadOnly
};
```

The `Vote` values are:

- `VoteCommit`. The Resource is able to write (or has already written) all the data needed to commit the transaction to stable storage, as well as an indication that it has prepared the transaction.
- `VoteRollback`. For any reason, the Resource could not vote to commit the transaction. This includes not having any knowledge about the transaction (which might happen after a crash).
- `VoteReadOnly`. No persistent data associated with the Resource has been modified by the transaction.

Structures

The `CosTransactions` module defines these structures, which are used to save the transaction context.

- `otid_t` contains an object transaction ID (or `otid`), which is a globally unique ID for a transaction. The `otid_t` structure is a more efficient OMG IDL version of the X/Open-defined transaction identifier (XID). The `otid_t` can be transformed to an X/Open XID and vice versa.
- `TransIdentity` contains certain key information for a transaction: its Coordinator, its Terminator (optionally), and its `otid`.
- `PropagationContext` contains a transaction's `TransIdentity` and its time-out. In addition, it contains a `TransIdentity` for the parent and each ancestor transaction, up to the top-level transaction, formatted as a sequence (or array). Because nested transactions are not implemented in `VisiTransact`, every transaction is a top-level transaction, and the `parents` sequence will always be empty.

For the most part, these structures are used behind the scenes; you won't reference them directly.

```
struct otid_t
{
    long formatID;
    long bqual_length;
    sequence <octet> tid;
};
struct TransIdentity
{
    Coordinator coordinator;
    Terminator terminator;
    otid_t otid;
};
struct PropagationContext
{
    unsigned long timeout;
    TransIdentity current;
    sequence <TransIdentity> parents;
    any implementation_specific_data;
};
```

When the transaction context is passed from one object to another object, usually a `TransactionalObject`, it is commonly passed as a `PropagationContext`. The `implementation_specific_data` field is reserved for the `VisiTransact` Transaction Service.

For the most part, these structures are used behind the scenes; you won't reference them directly. Certain methods, however, work explicitly with `PropagationContext`.

- `Coordinator::get_txcontext()` extracts a `PropagationContext`.
- `TransactionFactory::recreate()` uses a `PropagationContext` to create a new `Control` object.

The transaction context is always passed to a transactional object implicitly. In addition, a program may be passed a transaction context explicitly, as a parameter. You can use `Coordinator::get_txcontext()` to get the `PropagationContext`. For more information on propagation of transaction context, see [“TransactionalObject interface” on page 165](#).

Another method that obtains information from these structures is the `VISTransactions::Current::get_otid()` method, which extracts the `otid` from the `PropagationContext`.

Exceptions

Exceptions are divided into three categories: Standard, Heuristic, and Method-specific.

Table 10.1 Standard exceptions in the CosTransactions module

Exception	When this exception is thrown ...
CORBA::INVALID_TRANSACTION	The invoking thread has an invalid transaction context.
CORBA::NO_PERMISSION	The invoking thread does not have permission to complete the transaction. For example, only the transaction-originator thread can call this method.
CORBA::TRANSACTION_REQUIRED	The invoking thread does not have a transaction context.
CORBA::TRANSACTION_ROLLEDBACK	The transaction has been rolled back.
CORBA::WrongTransaction	Raised by the ORB when returning the response to a deferred synchronous request. This exception is raised only if the request was implicitly associated with a different transaction than the thread requesting the response through <code>Request::get_response()</code> or <code>ORB::get_next_response()</code> . See the VisiBroker section on Dynamic Invocation Interface (DII).

Table 10.2 Heuristic exceptions in the CosTransactions module

Exception	When this exception is thrown ...
CosTransactions::HeuristicCommit	The rollback operation on a Resource raises this exception to report that a heuristic decision was made, and that all relevant updates have been committed.
CosTransactions::HeuristicMixed	A heuristic decision was made when attempting to commit the transaction. Some relevant updates have been committed and others have been rolled back.
CosTransactions::HeuristicHazard	A heuristic decision may have been made when attempting to commit the transaction. The disposition of all relevant updates is not known. For those updates whose disposition is known, either all have been committed or all have been rolled back. (In other words, the <code>HeuristicMixed</code> exception takes priority over the <code>HeuristicHazard</code> exception.)
CosTransactions::HeuristicRollback	The commit operation on a Resource raises this exception to report that a heuristic decision was made and that all relevant updates have been rolled back.

Table 10.3 Method-specific exceptions in the CosTransactions module

Exception	When this exception is thrown
CosTransactions::Inactive	The transaction has already been prepared.
CosTransactions::InvalidControl	The <code>Control</code> parameter passed to resume is not valid in the current execution environment.
CosTransactions::NotPrepared	A commit has been issued but the Resource has not been prepared.
CosTransactions::NoTransaction	No transaction is associated with the client thread.
CosTransactions::NotSubtransaction	Not raised in VisiTransact.
CosTransactions::SubtransactionsUnavailable	Because VisiBroker VisiTransact does not support nested transactions, this exception is raised if an attempt is made to begin a transaction when a transaction is already in progress for this client thread.
CosTransactions::SynchronizationUnavailable	Not raised in VisiTransact.
CosTransactions::Unavailable	The requested object cannot be provided. For example, the <code>Control</code> object could not provide a Terminator.

Looking at the VISTransactions module


Interfaces in the `VISTransactions` module inherit from and extend the `CosTransactions` interfaces. The `VISTransactions` module defines no new data types, structures, or exceptions over those in `CosTransactions`. For example, the `Current` interface includes `VisiBroker VisiTransact` methods that make certain programming operations shorter and more convenient. The IDL for this module is contained in the file `VISTransactions.idl`.

For related information see “Choosing a Current interface” on page 151 and “Obtaining a Current object reference” on page 152.

Current interface

The `Current` interface defines methods to:

- Enable a program to manage transactions.
- Use implicit transaction propagation.
- Obtain information about the current transaction.
- Register Resources and Synchronization objects.

`VisiBroker VisiTransact` supports a number of extensions to the OMG Transaction Service specification—additional methods for added convenience. `VisiBroker VisiTransact` methods on the `Current` interface can simplify the use of the `VisiTransact Transaction Service` for most programs. These methods are flagged by the icon  where described or cross-referenced.

Choosing a Current interface

The `VisiTransact Transaction Service` provides the `Current` interface in the following IDL files:

- **`CosTransactions.idl`** contains the Transaction Service IDL that conforms to the final OMG Transaction Service document.
- **`VISTransactions.idl`** provides both the `CosTransactions` interface and the `VISTransactions` interface, which inherits and extends the `CosTransactions::Current` interface. This interface includes `VisiBroker VisiTransact` extensions such as `begin_with_name()`, `register_resource()`, and others.

You should use one of these IDL files. To restrict yourself strictly to CORBA-compliant methods, use the **`CosTransactions.idl`**. If you decide to use any of the `VisiTransact` extensions, use **`VISTransactions.idl`**.

The following example shows the `CosTransactions` interface for `Current`.

```

:
interface Current
{
    void begin()
        raises(SubtransactionsUnavailable);
    void commit(in boolean report_heuristics)
        raises (NoTransaction,
            HeuristicMixed,
            HeuristicHazard);
    void rollback()
        raises(NoTransaction);
    void rollback_only()
        raises(NoTransaction);
}

```

```

Status get_status();
string get_transaction_name();
void set_timeout(in unsigned long seconds);
Control get_control();
Control suspend();
void resume(in Control which)
    raises(InvalidControl);
};
:

```

The next example shows the `VIStransactions` interface for `Current`.

```

interface Current : CosTransactions::Current
{
    void begin_with_name(in string user_transaction_name)
        raises(CosTransactions::SubtransactionsUnavailable);
    CosTransactions::RecoveryCoordinator
        register_resource(in CosTransactions::Resource resource)
            raises(CosTransactions::Inactive);
    void register_synchronization(in CosTransactions::Synchronization synch)
        raises(CosTransactions::NoTransaction,
            CosTransactions::Inactive,
            CosTransactions::SynchronizationUnavailable,
            CosTransactions::Unavailable);
    CosTransactions::otid_t get_otid();
        raises(CosTransactions::NoTransaction,
            CosTransactions::Unavailable);
    CosTransactions::PropagationContext get_txcontext()
        raises(CosTransactions::Unavailable,
            CosTransactions::NoTransaction);
    attribute string ots_name;
    attribute string ots_host;
    attribute string ots_factory;
};

```

Obtaining a Current object reference

To gain access to a VisiTransact-managed transaction, you must obtain an object reference to the `Current` object. The `Current` object reference is valid throughout the process.

The example below shows how a reference to the `Current` object is obtained using the `resolve_initial_references()` method, and then how the object returned by that method is narrowed to a `CosTransactions::Current` object.

```

int main(...)
{
    try
    {
        :
        // ORB related initialization
        // get reference to a CosTransactions::Current instance
        CORBA::Object_var
            obj = orb->resolve_initial_references("TransactionCurrent");
        CosTransactions::Current_var
            current = CosTransactions::Current::_narrow(obj);
        :
    }
    catch(...) { } // catch all exceptions or exceptions you care about,
}

```


VisiBroker VisiTransact offers extensions to the `Current` interface to simplify certain operations. To take advantage of these extensions, narrow to a `VISTransactions::Current` object.

```
VISTransactions::Current::_narrow(obj)
```

Using the Current object reference

The `Current` object reference is valid for the entire process under which you create it; you can use it in any thread. You can either make multiple calls to obtain references to the `Current` object or use just one reference throughout the entire process. Typically, you would obtain one reference to avoid multiple invocations of `resolve_initial_references()`.

The C++ header files that you include must also correspond to your choice of interfaces.

- For `VISTransactions`, use `#include <VISTransactions_c.hh>`.
- For `CosTransactions`, use `#include <CosTransactions_c.hh>`.

For more information, see the *VisiTransact Guide*.

Is your VisiTransact Transaction Service instance available?

You can issue `begin()` or `begin_with_name()` to determine if the instance of your VisiTransact Transaction Service is available. The method will raise `CORBA:NO_IMPLEMENT` exception if the instance is not available.

Calling `get_status()` when there is no available instance of the VisiTransact Transaction Service will return the current transaction state, and cannot be used to determine if the instance of the VisiTransact Transaction Service is available.

Checked behavior

Checked behavior is supported by the VisiTransact Transaction Service to provide an extra level of transaction integrity. Specifically, checked behavior is supported for transactions originated with `Current::begin()`. The purpose of the checks is to ensure that all transactional requests made by the application have completed their processing before the transaction is committed. This guarantees that a commit will not succeed unless all transactional objects involved in the transaction have completed the processing of their transactional requests. For checks that are part of the commit process, see `commit()`. For more information about checked behavior, see the *VisiTransact Guide*.

Current methods

```
begin()
void begin()
raises SubtransactionsUnavailable;
```

This method creates a new transaction. Because nested transactions are *not* supported in VisiBroker VisiTransact, this is always a top-level transaction.

The transaction context of the client thread is modified so that the thread is associated with the new transaction. If the client thread is already associated with a transaction, the `SubtransactionsUnavailable` exception is raised.

Included in the `Current` interface in **CosTransactions.idl**

The following exceptions may be raised when calling this method.

Exception	When thrown
<code>CosTransactions::SubtransactionsUnavailable</code>	Because VisiBroker VisiTransact does not support nested transactions, this exception is raised if a transaction is already in progress for this client thread.

Related methods:

- `begin_with_name()`
- `commit()`
- `get_terminator()` in Control interface
- `rollback()`
- `rollback_only()`

For more information, see the *VisiTransact Guide*.

`begin_with_name()`

```
void begin_with_name(in string user_transaction_name)
raises(CosTransactions::SubtransactionsUnavailable);
```

- This VisiBroker VisiTransact method is a `begin()` method that enables its caller to pass a user-defined informational transaction name. For example, this helps with diagnostics because the user-defined transaction name is included in the value returned by the `get_transaction_name()` method. The name also helps with administration, because the Console will report the name in the detailed information about an outstanding transaction.

To use this method, narrow the object returned from `resolve_initial_references()` to `VISTransactions::Current`. For more information, see [“Obtaining a Current object reference” on page 152](#).

Included in the `Current` interface in **VISTransactions.idl**

The following parameters are used by this method.

Parameter	Description
<code>user_transaction_name</code>	This user-defined informational transaction name can be used to trace transactions and debug programs.

The following exceptions may be raised when calling this method.

Exception	When thrown
<code>CosTransactions::SubtransactionsUnavailable</code>	This exception is thrown if the thread already has a transaction context.

Related methods:

- `begin()`
- `commit()`
- `get_terminator()` in Control interface
- `rollback()`
- `rollback_only()`

```

commit()

void commit(in boolean report_heuristics)
raises(NoTransaction,
       HeuristicMixed,
       HeuristicHazard
);

```

This method commits the transaction associated with the client thread. The effect of this method is equivalent to calling the `commit()` method on the corresponding Terminator object.

If this transaction has been marked for rollback, or any Resource votes for rollback, this call raises `CORBA::TRANSACTION_ROLLEDBACK`. If there is no current transaction, a `CosTransactions::NoTransaction` exception is raised. If the caller is not the transaction originator, `commit()` raises the exception `CORBA::NO_PERMISSION`.

Checks are made to ensure checked behavior. See the VisiBroker *VisiTransact Guide* for more information.

On return from this method, the client thread is no longer associated with a transaction. Any attempt to use `Current`, as if there were a transaction, will raise an exception, such as `NoTransaction` or `CORBA::TRANSACTION_REQUIRED`, or will return a null object reference.

This method does not return until the transaction is complete, and all related Synchronization objects have been notified.

Included in the `Current` interface in **CosTransactions.idl**

The following parameters are used by this method.

Parameter	Description
<code>in boolean report_heuristics</code>	<p><code>true</code>—Requests that the program be notified when heuristic decisions are made.</p> <p><code>false</code>—Requests that the heuristic information is not returned to the program.</p>

The following exceptions may be raised when calling this method.

Exception	When thrown
<code>CosTransactions::NoTransaction</code>	No transaction is associated with the client thread.
<code>CosTransactions::HeuristicMixed</code>	A heuristic decision was made and <code>report_heuristics</code> is <code>true</code> . Some relevant updates have been committed and others have been rolled back.
<code>CosTransactions::HeuristicHazard</code>	A heuristic decision may have been made and <code>report_heuristics</code> is <code>true</code> . The disposition of all relevant updates is not known. For those updates whose disposition is known, either all have been committed or all have been rolled back. If the known updates are a mixture of commits and rollbacks, then the <code>HeuristicMixed</code> exception is raised.
<code>CORBA::NO_PERMISSION</code>	Only the transaction-originator thread can call this method.
<code>CORBA::OBJECT_NOT_EXIST</code>	It is unknown whether the transaction was committed or rolled back because a different thread or process could have terminated the transaction already. For example, the transaction has already timed out.
<code>CORBA::TRANSACTION_ROLLEDBACK</code>	The transaction was rolled back.

Related methods:

- **W** begin()
- begin_with_name()
- commit() in Terminator interface
- get_terminator() in Control interface
- resume()
- rollback()

For more information on the heuristic log, see the *VisiTransact Guide*.

`get_control()`

```
Control get_control();
```

This method returns a Control object reference that represents the transaction context currently associated with the client thread.

If the client thread is not associated with a transaction, a null object reference is returned.

Caution See the VisiBroker *VisiTransact Guide* for details on checked behavior and the implications of using this method.

Included in the `Current` interface in **CosTransactions.idl**.

No user exceptions are raised.

Related methods:

- resume()
- suspend()

For related material, see [“Control interface” on page 168](#) and [“Terminator interface” on page 170](#). For more information see “VisiTransact basics” in the *VisiTransact Guide*.

`get_otid()`

```
CosTransactions::otid_t get_otid()
raises(CosTransactions::NoTransaction,
       CosTransactions::Unavailable);
```

W Most applications will not normally call this method.

This `VisiTransactions::Current` method provides the object transaction ID (`otid`) through the `Current` interface as a convenience. This avoids going to the Coordinator and looking through a `PropagationContext`. The `otid` is used to identify a transaction to a recoverable object. This method raises `CosTransactions::NoTransaction` if no transaction is associated with the client thread.

To use this method, narrow the object returned from `resolve_initial_references()` to `VisiTransactions::Current`. For more information, see [“Obtaining a Current object reference” on page 152](#).

Included in the `Current` interface in **VISTransactions.idl**.

The following exceptions may be raised when calling this method.

Exception	When thrown
<code>CosTransactions::NoTransaction</code>	No transaction is associated with the client thread.
<code>CosTransactions::Unavailable</code>	This exception is thrown if the VisiTransact Transaction Service chooses to restrict the availability of the <code>PropagationContext</code> .

Related methods:

- `get_txcontext()`
- `get_txcontext()` in Coordinator interface
- `get_control()`

For related material, see [“Coordinator interface” on page 172](#) and [“Terminator interface” on page 170](#).

`get_status()`

```
Status get_status();
```

This method returns an enumerated value (`enum Status`) that represents the status of the transaction associated with the client thread.

Calling this method is equivalent to calling the `get_status()` method on the corresponding Coordinator object. If there is no transaction associated with the current thread, then the method returns `CosTransactions::StatusNoTransaction`.

The possible return values are:

- `StatusActive`
- `StatusMarkedRollback`
- `StatusPrepared`
- `StatusCommitted`
- `StatusRolledBack`
- `StatusUnknown`
- `StatusNoTransaction`
- `StatusPreparing`
- `StatusCommitting`
- `StatusRollingBack`

Included in the `Current` interface in **`CosTransactions.idl`**.

No user exceptions are raised.

Status value definitions

Some implications of the `enum Status` values are:

- `StatusActive`—A transaction is associated with the target object and it is in the active state. The VisiTransact Transaction Service returns this status after a transaction has been started and prior to a Coordinator issuing any prepare statements—unless the transaction has been marked for rollback or timed out.
- `StatusMarkedRollback`—A transaction is associated with the target object and has been marked for rollback, perhaps as the result of the `rollback_only()` method.
- `StatusPrepared`—A transaction is associated with the target object and has been prepared.
- `StatusCommitted`—A transaction is associated with the target object and has been committed. It is likely that heuristics exist, otherwise the transaction would have been quickly destroyed and `StatusNoTransaction` returned.
- `StatusRolledBack`—A transaction is associated with the target object and the outcome has been determined as rollback. It is likely that heuristics exist, otherwise the transaction would have been quickly destroyed and `StatusNoTransaction` returned.
- `StatusUnknown`—A transaction is associated with the target object, but the VisiTransact Transaction Service cannot determine its current status. This is a transient condition, and a subsequent invocation will ultimately return a different status.
- `StatusNoTransaction`—No transaction is currently associated with the target object. This will occur after a transaction has completed.

- `StatusPreparing`—A transaction is associated with the target object and it is in the process of preparing. The `VisiTransact Transaction Service` returns this status if the transaction has started preparing, but has not yet completed the process—perhaps because it is waiting for responses to prepare from one or more `Resources`.
- `StatusCommitting`—A transaction is associated with the target object and is in the process of committing. The `VisiTransact Transaction Service` returns this status if the transaction has begun to commit, but has not yet completed the process—perhaps because it is waiting for responses from one or more `Resources`.
- `StatusRollingBack`—A transaction is associated with the target object and it is in the process of rolling back. The `VisiTransact Transaction Service` returns this status if the transaction is being rolled back, but has not yet completed the process—perhaps because it is waiting for responses from one or more `Resources`.

Related methods:

- `get_status()` in `Coordinator` interface

`get_transaction_name()`

```
string get_transaction_name();
```

This method returns a printable string that is a descriptive name for the transaction. This method is intended to assist in diagnostics and debugging. If the transaction was created by the `begin_with_name()` method, the returned string is the user-defined name assigned to the transaction, rather than the `VisiTransact Transaction Service`-generated name.

The effect of this method is equivalent to calling the `get_transaction_name()` method on the corresponding `Coordinator` object. If there is no transaction associated with the client thread, an empty string is returned.

Included in the `Current` interface in **`CosTransactions.idl`**.

No user exceptions are raised.

Related methods:

- `begin_with_name()`
- `create_with_name()` in `TransactionFactory` interface
- `get_transaction_name()` in `Coordinator` interface

`get_txcontext()`

```
CosTransactions::PropagationContext get_txcontext()
raises(CosTransactions::Unavailable,
       CosTransactions::NoTransaction);
```

- Most applications will not normally call this method.

This `VISTransactions::Current` method returns a `PropagationContext`, which can be used by one `VisiTransact Transaction Service` domain to export a transaction to a new `VisiTransact Transaction Service` domain.

To use this method, narrow the object returned from `resolve_initial_references()` to `VISTransactions::Current`. For more information, see [“Obtaining a Current object reference” on page 152](#).

Included in the `Current` interface in **`VISTransactions.idl`**.

The following exceptions may be raised when calling this method.

Exception	When thrown
<code>CosTransactions::Unavailable</code>	This exception is thrown if the <code>VisiTransact Transaction Service</code> chooses to restrict the availability of the <code>PropagationContext</code> .
<code>CosTransactions::NoTransaction</code>	No transaction is associated with the client thread.

Related methods:

- `get_txcontext()` in `Coordinator` interface
- `Coordinator` interface

For related material, see [“Coordinator interface” on page 172](#) and [“Terminator interface” on page 170](#).

`ots_factory`

```
attribute string ots_factory;
```

W If you are using **VISTransactions.idl**, you can control the instance of the `VisiTransact Transaction Service` used to create a transaction by setting this attribute before you call `VISTransactions::Current::begin()`. Subsequent calls to the `begin()` method create transactions on the specified `VisiTransact Transaction Service`. This attribute applies to all the threads in your program. When the attribute is set, it retains its value until set again.

This attribute specifies the `VisiTransact Transaction Service` instance by IOR. `VisiTransact` uses the specified IOR (`CosTransactions::TransactionFactory`) to locate the desired instance of a `VisiTransact Transaction Service` instance on the network. This argument enables `VisiTransact` to operate without the use of a Smart Agent (**osagent**).

If you specify the IOR with either the Host Name or `VisiTransact Transaction Service Name` attributes, the Smart Agent will find the `VisiTransact Transaction Service` instance by IOR only—it ignores the other attributes. If you leave all three attributes null, the ORB chooses a `VisiTransact Transaction Service` instance using the `VisiBroker` Smart Agent.

Included in the `Current` interface in **VISTransactions.idl**.

To set this attribute, use the appropriate method generated automatically for the language you are using.

Related attributes:

- W** ▪ `ots_host`
- W** ▪ `ots_name`

For more information, see the `VisiBroker VisiTransact Guide`.

`ots_host`

```
attribute string ots_host;
```

W If you are using **VISTransactions.idl**, you can control the instance of the `VisiTransact Transaction Service` used to create a transaction by setting this attribute before you call `VISTransactions::Current::begin()`. Subsequent calls to the `begin()` method create transactions on the specified `VisiTransact Transaction Service`. This attribute applies to all the threads in your program. When the attribute is set, it retains its value until set again. To return this attribute to the default `VisiTransact` instance, set it to an empty or null string.

This attribute specifies the VisiTransact Transaction Service instance by host name. The Smart Agent will find any available VisiTransact Transaction Service instance that is located on the specified host.

If you specify a combination of Host Name and VisiTransact Transaction Service Name attributes, the Smart Agent will find the named VisiTransact Transaction Service instance on the named host. If you leave all three attributes null, the ORB chooses a VisiTransact Transaction Service instance using the VisiBroker Smart Agent.

Included in the `Current` interface in **VISTransactions.idl**.

To set this attribute, use the appropriate method generated automatically for the language you are using.

Related attributes:

- `ots_factory`
- `ots_name`

For more information, see the VisiBroker *VisiTransact Guide*.

`ots_name`

```
attribute string ots_name;
```

- If you are using **VISTransactions.idl**, you can control the instance of the VisiTransact Transaction Service used to create a transaction by setting this attribute before you call `VISTransactions::Current::begin()`. Subsequent calls to the `begin()` method create transactions on the specified VisiTransact Transaction Service. This attribute applies to all the threads in your program. When the attribute is set, it retains its value until set again. To return this attribute to the default VisiTransact instance, set it to an empty or null string.

This attribute specifies the VisiTransact Transaction Service instance by name. The Smart Agent will find the named VisiTransact Transaction Service instance anywhere on the network.

If you specify a combination of Host Name and VisiTransact Transaction Service Name attributes, the Smart Agent will find the named VisiTransact Transaction Service instance on the named host. If you leave all three attributes null, the ORB chooses a VisiTransact Transaction Service instance using the VisiBroker Smart Agent.

Included in the `Current` interface in **VISTransactions.idl**.

To set this attribute, use the appropriate method generated automatically for the language you are using.

Related attributes:

- `ots_factory`
- `ots_host`

For more information, see the VisiBroker *VisiTransact Guide*.

register_resource()

```

CosTransactions::RecoveryCoordinator
    register_resource(in CosTransactions::Resource resource)
raises(CosTransactions::Inactive);

```

W Most applications will not normally call this method.

This `VISTransactions::Current` method registers a `Resource` for a recoverable object. This method is a shortcut for using the `Control` and `Coordinator` objects to register a `Resource` for a recoverable object. It returns a `Recovery Coordinator` object that can be used to help coordinate recovery. If this method is invoked when there is no transaction associated with the client thread, the `CORBA::TRANSACTION_REQUIRED` exception is thrown.

To use this method, narrow the object returned from `resolve_initial_references()` to `VISTransactions::Current`. For more information, see [“Obtaining a Current object reference” on page 152](#).

Included in the `Current` interface in **VISTransactions.idl**.

The following parameters are used by this method.

Parameter	Description
<code>in CosTransactions::Resource resource</code>	The <code>Resource</code> object for the recoverable object.

The following exceptions may be raised when calling this method.

Exception	When thrown
<code>CosTransactions::Inactive</code>	The transaction has already been prepared.
<code>CORBA::TRANSACTION_ROLLEDBACK</code>	The transaction has been marked for rollback.

Related methods:

- `register_resource()` in `Coordinator` interface
- `get_control()`

For related material, see [“Coordinator interface” on page 172](#).

register_synchronization()

```

void register_synchronization(in CosTransactions::Synchronization synch)
raises(CosTransactions::NoTransaction,
      CosTransactions::Inactive,
      CosTransactions::SynchronizationUnavailable,
      CosTransactions::Unavailable);

```

W This `VISTransactions::Current` method registers a `Synchronization` object. This method is a short-cut for using the `Control` and `Coordinator` object to register a `Synchronization` object. To use this method, narrow the object returned from `resolve_initial_references()` to `VISTransactions::Current`. For more information, see [“Obtaining a Current object reference” on page 152](#).

Included in the `Current` interface in **VISTransactions.idl**.

The following parameters are used by this method.

Parameter	Description
<code>in CosTransactions::Synchronization synch</code>	The <code>Synchronization</code> object to register.

The following exceptions may be raised when calling this method.

Exception	When thrown
<code>CosTransactions::NoTransaction</code>	No transaction is associated with the client thread.
<code>CosTransactions::Inactive</code>	The transaction has already been prepared.
<code>CosTransactions::SynchronizationUnavailable</code>	This exception is not raised by <code>VisiBroker VisiTransact</code> .
<code>CosTransactions::Unavailable</code>	Raised if the <code>VisiTransact Transaction Service</code> restricts the availability of the <code>PropagationContext</code> .

Related methods:

- `register_synchronization()` in `Coordinator` interface

For more information, see the *VisiTransact Guide*.

`resume()`

```
void resume(in Control which)
raises(InvalidControl);
```

Associates the client thread with the specified transaction. Typically, this is used to either

- Associate a transaction context with a thread for use in implicit transaction propagation, or
- Resume a transaction that was previously suspended by a `suspend()` method.

The client thread becomes associated with the specified transaction. If the client thread was already associated with a transaction, the previous transaction context is forgotten. If `resume()` is invoked with a `NULL` control, no transaction is associated with the current thread, and the transaction context is forgotten.

Caution Any transaction context you set via `resume()` is propagated back to the invoking object.

Included in the `Current` interface in **`CosTransactions.idl`**.

The following parameters are used by this method.

Parameter	Description
<code>in Control which</code>	A <code>Control</code> object used to set the thread's transaction context.

The following exceptions may be raised when calling this method.

Exception	When thrown
<code>CosTransactions::InvalidControl</code>	The <code>Control</code> parameter passed to <code>resume</code> is not valid in the current execution environment.

Related methods:

- `get_control()`
- `suspend()`

For more information, see the `VisiBroker VisiTransact Guide`.

`rollback()`

```
void rollback()
raises(NoTransaction);
```

Rolls back the transaction associated with the client thread. This is equivalent to calling the `rollback()` method on the corresponding Terminator object. This method does not return until the transaction is complete, and all related Synchronization objects have been notified. On return from this method, the client thread is no longer associated with a transaction. Any attempt to use Current, as if there were a transaction, will raise an exception, such as `CosTransactions::NoTransaction` or `CORBA::TRANSACTION_REQUIRED`, or return a null object reference. If a heuristic occurs, this method will not throw a heuristic-related exception.

If the caller is not the transaction originator, `rollback()` raises the exception `CORBA::NO_PERMISSION`.

Included in the Current interface in **CosTransactions.idl**.

The following exceptions may be raised when calling this method.

Exception	When thrown
<code>CosTransactions::NoTransaction</code>	No transaction is associated with the current client thread.
<code>CORBA::NO_PERMISSION</code>	Only the transaction-originator thread can call this method.
<code>CORBA::OBJECT_NOT_EXIST</code>	It is unknown whether the transaction was committed or rolled back because a different thread or process could have terminated the transaction already. For example, the transaction has already timed out.

Related methods:

- `commit()`
- `rollback()` in Terminator interface
- `rollback_only()`

For more information, see the VisiBroker *VisiTransact Guide*.

`rollback_only()`

```
void rollback_only()
raises(NoTransaction);
```

The method modifies the transaction associated with the client thread so that `rollback` is the only possible transaction outcome. The effect of this request is equivalent to calling the `rollback_only()` method on the corresponding Coordinator object. A client that is restricted from performing the `rollback()` operation, can nonetheless call `rollback_only()`.

Included in the Current interface in **CosTransactions.idl**.

Exceptions

The following exceptions may be raised when calling this method.

Exception	When thrown
<code>CosTransactions::NoTransaction</code>	No transaction is associated with the current client thread.

Related methods:

- `rollback()`
- `rollback_only()` of Coordinator interface

For more information, see the VisiBroker *VisiTransact Guide*.

```
set_timeout()
```

```
void set_timeout(in unsigned long seconds);
```

This method establishes a new timeout for transactions started by subsequent calls to the `Current::begin()` method in all threads within this program.

To establish a new timeout, use these values of the `seconds` parameter:

- `= 0`—Sets any subsequent transaction that is begun to the default transaction timeout for the VisiTransact Transaction Service instance that it uses.
- `> 0`—Sets the new timeout to the specified number of seconds. If the `seconds` parameter exceeds the maximum timeout valid for a VisiTransact Transaction Service instance being used, then the new timeout is set to that maximum, to bring it in range.

Note When a transaction, created by a subsequent call to `begin()` in any thread in the process, takes longer to start transaction completion than the established timeout, it will be rolled back. If the timeout occurs before the transaction enters the completion stage (begins two-phase or one-phase processing) the transaction will be rolled back. Otherwise, the timeout is ignored.

Included in the `Current` interface in **CosTransactions.idl**.

The following parameters are used by this method.

Parameter	Description
<code>in unsigned long seconds</code>	Numbers of seconds before timeout will occur on subsequent <code>begin()</code> operations.

No user exceptions are raised.

Related methods:

For other methods that affect transaction timeout, see:

- `create()` in TransactionFactory interface
- `create_with_name()` in TransactionFactory interface

For more information see the description of `set_timeout()` in the *VisiTransact Guide*.

```
suspend()
```

```
Control suspend();
```

This method suspends the transaction currently associated with the client thread and returns a `Control` object for that transaction. If the client thread is not associated with a transaction, a null object reference is returned.

The `Control` object can be passed to the `resume()` method to reestablish this context in the same thread or a different thread.

After the call to `suspend()`, no transaction is associated with the client thread. Any attempt to use `Current`, as if there were a transaction, will raise an exception, such as `CosTransactions::NoTransaction` or `CORBA::TRANSACTION_REQUIRED`, or return a null object reference.

Included in the `Current` interface in **CosTransactions.idl**.

No user exceptions are raised.

Related methods:

- `get_control()`
- `resume()`

TransactionalObject interface

The `TransactionalObject` interface provides for the automatic propagation of transaction context on method calls of transactional objects. The `TransactionalObject` interface defines no methods.

Methods that work on transactions must have access to the transaction context. The transaction context can be made available to such methods in two ways:

- **Explicit propagation.** A method receives and passes the transaction context as a Terminator, Control, Coordinator, or PropagationContext structure. For further information, see the VisiBroker *VisiTransact Guide*.
- **Implicit propagation.** The transaction context is passed automatically (and implicitly) on method calls. For further information, see the VisiBroker *VisiTransact Guide*.

Implicit propagation is the typical, and easiest, way. This is the capability that the `TransactionalObject` interface provides to your transactional objects.

For information about the details of what information is in the transaction context, see [“Structures” on page 149](#).

An instance of `TransactionalObject` can participate in implicit propagation. Implicit propagation is where the transaction context associated with the client thread is automatically propagated to `TransactionalObject` instances through method calls.

To use VisiTransact-managed transactions, all of your transactional objects must inherit from `TransactionalObject`. By using VisiTransact-managed transactions, you benefit from checked behavior.

The following example shows the `TransactionalObject` interface in the `CosTransactions.idl` file.

```
interface TransactionalObject
{
};
```

The transaction context is always passed implicitly to an object that inherits from `CosTransactions::TransactionalObject`. In addition, a program may be passed a transaction context explicitly, as a parameter.

TransactionFactory interface

As described in [“Current interface” on page 151](#) the `Current` interface enables a program to initiate VisiTransact-managed transactions. This section, by contrast, describes the `TransactionFactory` interface, which defines methods that enable a program to initiate non-VisiTransact-managed transactions. The `TransactionFactory` interface gives programs direct control over the propagation of transaction context.

In the `CosTransactions` module, the `TransactionFactory` interface provides three methods:

- `create()`—Begins a transaction.
- `create_with_name()`—Available if you are using the `VISTransactions` IDL interface (with the VisiBroker VisiTransact extensions).
- `recreate()`—Creates a new representation of a transaction.

For further information about using different IDL files, see [“Choosing a Current interface” on page 151](#).

Note You acquire a `TransactionFactory` object the way you do any CORBA object; for example, by binding.

Methods that are VisiBroker VisiTransact extensions are flagged by the icon **W** where described or cross-referenced.

The following example shows the `CosTransactions` IDL for `TransactionFactory`.

```

:
interface TransactionFactory
{
    Control create(in unsigned long time_out);
    Control recreate(in PropagationContext ctx);
};
:

```

The next example shows the `VISTransactions` IDL for `TransactionFactory`.

```

:
interface TransactionFactory : CosTransactions::TransactionFactory
{
    CosTransactions::Control
        create_with_name(in unsigned long time_out,
                        in string user_transaction_name);
};...
:

```

TransactionFactory methods

`create()`

```
CosTransactions::Control create(in unsigned long time_out);
```

This method accepts a timeout parameter (`time_out`) and creates a new transaction. It returns a `Control` object. The `Control` object can be used to manage or to control participation in the new transaction. The `Control` object can be used by any thread and passed around explicitly, just like any other CORBA object.

Note Checked behavior cannot be provided for transactions that use this method.

Included in the `TransactionFactory` interface in **CosTransactions.idl**.

The following parameters are used by this method.

Parameter	Description
<code>in unsigned long time_out</code>	A timeout, in seconds, that applies to this invocation only.

To establish a new timeout, use the following values of the `time_out` parameter.

- **= 0**—Sets any subsequent transaction that is begun to the default transaction timeout for the VisiTransact Transaction Service instance that it uses.
- **> 0**—Sets the new timeout to the specified number of seconds. If the `seconds` parameter exceeds the maximum timeout valid for a VisiTransact Transaction Service instance being used, then the new timeout is set to that maximum.

Note If a transaction does not start transaction completion (begin two-phase or one-phase processing) before the timeout expires, it will be rolled back.

The new timeout applies only to the transaction created on this call.

See the description of `set_timeout()` in the VisiBroker *VisiTransact Guide*.

No user exceptions are raised.

Related methods:

- W** ▪ `create_with_name()`
- `get_terminator()` in `Control` interface
- `get_coordinator()` in `Control` interface

```
create_with_name()
```

```
CosTransactions::Control
create_with_name(in unsigned long time_out,
                 in string user_transaction_name);
```

W This `VISTransactions` method extends the `CosTransactions::TransactionFactory::create()` method by enabling you to create a new transaction and assign it an informational transaction name that can be used for debugging and error reporting. The user-defined transaction name is included in the value returned by `get_transaction_name()`.

Note Checked behavior cannot be provided for transactions that use this method.

This method returns a `Control` object. The `Control` object can be used to manage or to control participation in the new transaction. The `Control` object can be used by any thread and passed around explicitly, just like any other CORBA object.

Included in the `TransactionFactory` interface in **`VISTransactions.idl`**.

The following parameters are used by this method.

Parameter	Description
<code>in unsigned long time_out</code>	A timeout, in seconds, for this transaction.
<code>in string user_transaction_name</code>	This user-defined informational transaction name can be used to trace transactions and debug applications.

To establish a new timeout, use these values of the `time_out` parameter:

- `= 0`—Sets any subsequent transaction that is begun to the default transaction timeout for the `VisiTransact` Transaction Service instance that it uses.
- `> 0`—Sets the new timeout to the specified number of seconds. If the `seconds` parameter exceeds the maximum timeout valid for `VisiTransact` Transaction Service instance being used, then the new timeout is set to that maximum.

Note If a transaction does not start transaction completion (begin two-phase or one-phase processing) before the timeout expires, it will be rolled back.

The new timeout applies only to the transaction created on this call.

See the description of `set_timeout()` in the `VisiBroker VisiTransact Guide`.

No user exceptions are raised.

Related methods:

- `create()`
- `get_terminator()` in `Control` interface
- `get_coordinator()` in `Control` interface

For more information, see the `VisiBroker VisiTransact Guide`.

```
recreate()
```

```
Control recreate(in PropagationContext context);
```

Most applications will not normally call this method.

This method creates a new `Control` object using its `PropagationContext` parameter. The `Control` object can be used to manage or to control participation in the transaction.

To get a transaction's `PropagationContext`, invoke the `get_txcontext()` method on the transaction's `Coordinator` object.

Included in the `TransactionFactory` interface in **`CosTransactions.idl`**.

The following parameters are used by this method.

Parameter	Description
in <code>PropagationContext context</code>	Context of the transaction to import.

No user exceptions are raised.

The following example shows the way `get_txcontext()` and `recreate()` work together to recreate a transaction.

```

:
CosTransactions::Coordinator_var coord;
CosTransactions::TransactionFactory_var newDomainFactory;
:
propctxt = coord->get_txcontext();
CosTransactions::Control_var control = newDomainFactory->recreate(propctxt);
:

```

Related methods:

- `get_txcontext()` in `Current` interface
- `get_txcontext()` in `Coordinator` interface

Control interface

The `Control` interface enables a program to explicitly manage or propagate a transaction context. A `Control` object is implicitly associated with one specific transaction.

The `Control` interface defines two methods:

- `get_coordinator()`
- `get_terminator()`

The `get_coordinator()` method returns a `Coordinator` object, which supports methods used by participants in the transaction. The `get_terminator()` method returns a `Terminator` object, which supports methods to complete the transaction. The `Terminator` and `Coordinator` objects support methods that are typically performed by different parties. Providing two objects enables each set of methods to be made available only to the parties that require those methods.

The example below contains the IDL for the `Control` interface, an excerpt from the `CosTransactions.idl` file.

```

:
interface Control
{
    Terminator get_terminator()
        raises(Unavailable);
    Coordinator get_coordinator()
        raises(Unavailable);
};
:

```

You can obtain a `Control` object by using one of the methods of the `TransactionFactory`. See [“TransactionFactory interface” on page 165](#). You can also obtain a `Control` object for the current transaction (associated with a thread) through methods of the `Current` object. See `get_control()` or `suspend()` in `Current` interface.

Control methods

`get_coordinator()`

```
Coordinator get_coordinator()
raises(Unavailable);
```

This method returns a Coordinator object. The Coordinator provides methods that are called by participants in a transaction. These participants are typically either recoverable objects or agents of recoverable objects.

Included in the `Control` interface in **CosTransactions.idl**.

The following exceptions may be raised when calling this method.

Exception	When thrown
<code>CosTransactions::Unavailable</code>	The Control object cannot provide the requested Coordinator object.

Related methods:

- `get_control()` in Current interface

For more information, see [“Coordinator interface” on page 172](#) for details on methods you can use once you obtain the Coordinator object.

`get_terminator()`

```
Terminator get_terminator()
raises(Unavailable);
```

This method returns a Terminator object. The Terminator can be used to rollback or commit the transaction associated with the Control. The `Unavailable` exception is raised if the Control cannot provide the requested object due to the inability of the Terminator object to be transmitted to or be used in other execution environments.

Included in the `Control` interface in **CosTransactions.idl**.

The following exceptions may be raised when calling this method.

Exception	When thrown
<code>CosTransactions::Unavailable</code>	The Control object cannot provide the requested Terminator object.

Related methods:

- `get_control()` in Current interface
- `get_coordinator()`

For more information, see the VisiBroker *VisiTransact Guide*.

Terminator interface

The `Terminator` interface supports methods to commit or rollback a transaction. Typically, these methods are used by the transaction originator, but any program that has access to a `Terminator` object for that transaction can commit or rollback the transaction.

The following example contains the IDL for the `Terminator` interface, an excerpt from the `CosTransactions.idl` file.

```

:
interface Terminator
{
    void commit(in boolean report_heuristics)
        raises (HeuristicMixed,
              HeuristicHazard);
    void rollback();
};
:

```

Terminator methods

`commit()`

```

void commit(in boolean report_heuristics)
raises(HeuristicMixed,
      HeuristicHazard);

```

Before committing the transaction, this method performs some checks. If the transaction has not been marked rollback only, and all of the participants in the transaction agree to commit, the transaction is committed and the operation terminates normally. Otherwise, the transaction is rolled back and the `CORBA::TRANSACTION_ROLLEDBACK` standard exception is raised.

If the `report_heuristics` parameter is true, the `VisiTransact` Transaction Service will report inconsistent or possibly inconsistent outcomes using the `CosTransactions::HeuristicMixed` and `CosTransactions::HeuristicHazard` exceptions when appropriate. Information about the Resources involved in a heuristic outcome will be written to a heuristic log file corresponding to the instance of the `VisiTransact` Transaction Service. For more information on heuristics, see the *VisiTransact Guide*.

When a transaction is committed, all changes to recoverable objects made in the scope of this transaction are made permanent and visible to other transactions or clients.

Included in the `Terminator` interface in `CosTransactions.idl`.

The following parameters are used by this method.

Parameter	Description
<code>report_heuristics</code>	<p><code>true</code>—Requests that the <code>HeuristicMixed</code> or <code>HeuristicHazard</code> exceptions be raised, when appropriate.</p> <p><code>false</code>—Requests that the heuristic information is not returned to the program.</p>

The following exceptions may be raised when calling this method.

Exception	When thrown
<code>CosTransactions::HeuristicMixed</code>	A heuristic decision was made. Some relevant updates have been committed, and others have been rolled back.
<code>CosTransactions::HeuristicHazard</code>	A heuristic decision may have been made, the disposition of all relevant updates is not known. For those updates whose disposition is known, either all have been committed or all have been rolled back. If the known updates are a mixture of commits and rollbacks, then the <code>HeuristicMixed</code> exception is raised.
<code>CORBA::TRANSACTION_ROLLEDBACK</code>	The transaction has been marked for rollback.
<code>CORBA::OBJECT_NOT_EXIST</code>	It is unknown whether the transaction was committed or rolled back because a different thread or process could have terminated the transaction already. For example, the transaction has already timed out.

The example below shows how to use the `commit()` method with and without heuristics.

```
// Using commit() without heuristics.
try
{
    terminator->commit(0);
}
catch(CORBA::TRANSACTION_ROLLEDBACK&)
{
    cerr << "Transaction failed" << endl;
}
:
// Using commit() with heuristics.
try
{
    terminator->commit(1);
}
catch(CORBA::TRANSACTION_ROLLEDBACK&)
{
    cerr << "Transaction failed" << endl;
}
catch(CosTransactions::HeuristicMixed&)
{
    cerr << "HeuristicMixed exception was raised" << endl;
}
catch(CosTransactions::HeuristicHazard&)
{
    cerr << "HeuristicHazard exception was raised" << endl;
}
catch(CORBA::OBJECT_NOT_EXIST&)
{
    cerr << "Transaction no longer exists" << endl;
}
}
```

Related methods:

- `commit()` in `Current` interface
- `rollback()`

For more information, see the `VisiBroker VisiTransact Guide`.

`rollback()`

`void rollback();`

This method rolls back the transaction. When a transaction is rolled back, all changes to recoverable objects made in the scope of this transaction are rolled back. All Resources locked by the transaction are made available to other transactions as appropriate to the degree of isolation enforced by the Resources.

This method does not return until the transaction is complete and all related Synchronization objects have been notified. Any heuristic outcome that may occur will be provided through the Console.

Included in the `Terminator` interface in **CosTransactions.idl**.

The following exceptions may be raised when calling this method.

Exception	When thrown
<code>CORBA::NO_PERMISSION</code>	Only the transaction-originator thread can call this method.
<code>CORBA::OBJECT_NOT_EXIST</code>	It is unknown whether the transaction was committed or rolled back because a different thread or process could have terminated the transaction already. For example, the transaction has already timed out.

Related methods:

- `commit()`
- `rollback()` in `Current` interface
- `rollback_only()` in `Coordinator` interface

For more information, see the *VisiTransact Guide*.

Coordinator interface

The `Coordinator` interface provides methods that are used by participants in a transaction. These participants are typically either recoverable objects or agents of recoverable objects. Each Coordinator is implicitly associated with a single transaction.

The following example shows the `CosTransactions` IDL for the `Coordinator` interface.

```

:
interface Coordinator
{
    Status get_status();
    Status get_parent_status();
    Status get_top_level_status();

    boolean is_same_transaction(in Coordinator coord);
    boolean is_related_transaction(in Coordinator coord);
    boolean is_ancestor_transaction(in Coordinator coord);
    boolean is_descendant_transaction(in Coordinator coord);
    boolean is_top_level_transaction();

    unsigned long hash_transaction();
    unsigned long hash_top_level_tran();

    RecoveryCoordinator register_resource(in Resource resource)
        raises(Inactive);

    void register_synchronization(in Synchronization synch)
        raises(Inactive, SynchronizationUnavailable);

```

```

void register_subtran_aware(in SubtransactionAwareResource resource)
    raises(Inactive, NotSubtransaction);

void rollback_only()
    raises(Inactive);

string get_transaction_name();

Control create_subtransaction()
    raises(SubtransactionsUnavailable, Inactive);

PropagationContext get_txcontext()
    raises(Unavailable);
};

```

Because VisiTransact does not support nested transactions, several of the `Coordinator` methods have become equivalent—that is, they return the same result. More information is provided later in the section with the method descriptions.

The following methods are equivalent:

- `get_status()`
- `get_top_level_status()`
- `get_parent_status()`

Similarly, certain methods return true only when the target object and the parameter refer to the same `Coordinator` object. Therefore, the following methods are also equivalent:

- `is_same_transaction()`
- `is_related_transaction()`
- `is_ancestor_transaction()`
- `is_descendant_transaction()`

And, the following methods are equivalent:

- `hash_transaction()`
- `hash_top_level_tran()`

Finally, without nested transactions, the `create_subtransaction()` method is not useful and is therefore excluded from this version of VisiTransact (and this documentation), although it is described in the OMG specification.

Coordinator methods

`get_parent_status()`

Status `get_parent_status()`;

Because VisiTransact does not support nested transactions, every transaction is top-level, and `get_parent_status()` of a top-level transaction is equivalent to `get_status()`, by OMG definition. For further information, see `get_status()`.

Included in the `Coordinator` interface in **CosTransactions.idl**.

Related methods:

- `get_status()`

```
get_status()
```

```
Status get_status();
```

This method returns the status of the transaction associated with the target object, as an enumerated value (enum `Status`). If there is no transaction associated with the target object, then the method returns the value `StatusNoTransaction`.

The following are the possible return values, as defined in **CosTransactions.idl**:

- `StatusActive`
- `StatusMarkedRollback`
- `StatusPrepared`
- `StatusCommitted`
- `StatusRolledBack`
- `StatusUnknown`
- `StatusNoTransaction`
- `StatusPreparing`
- `StatusCommitting`
- `StatusRollingBack`

For information about each `Status` value, see “[Status value definitions](#)” on page 157.

Included in the `Coordinator` interface in **CosTransactions.idl**.

Related methods:

- `get_parent_status()`
- `get_status()` in `Current` interface
- `get_top_level_status()`

For more information, see the *VisiTransact Guide*.

```
get_top_level_status()
```

```
Status get_top_level_status();
```

Because `VisiTransact` does not support nested transactions, every transaction is top-level. Therefore, this method is equivalent to the `get_status()` method. See `get_status()`.

Included in the `Coordinator` interface in **CosTransactions.idl**.

Related methods:

- `get_status()`

```
get_transaction_name()
```

```
string get_transaction_name();
```

This method returns a printable string that is a descriptive name for the transaction. This method is intended to assist with diagnostics and debugging. If the transaction was created by the `VISTransactions::TransactionFactory::create_with_name()` method, the return string is the user-defined descriptive transaction name rather than the `VisiTransact` Transaction Service-generated name. If there is no transaction associated with the client thread, an empty string is returned.

Included in the `Coordinator` interface in **CosTransactions.idl**.

Related methods:

- `begin_with_name()` in `Current` interface
- `create_with_name()` in `TransactionFactory` interface
- `get_transaction_name()` in `Current` interface

For more information, see the *VisiTransact Guide*.

```
get_txcontext ()L
```

```
PropagationContext get_txcontext ()
raises(Unavailable);
```

Most applications will not normally call this method.

The `get_txcontext()` method returns a `PropagationContext`, which can be used by one `VisiTransact Transaction Service` domain to export a transaction to a new `VisiTransact Transaction Service` domain.

Included in the `Coordinator` interface in **CosTransactions.idl**.

The following exceptions may be raised when calling this method.

Exception	When thrown
<code>CosTransactions::Unavailable</code>	The <code>VisiTransact Transaction Service</code> restricts the availability of the <code>PropagationContext</code> .

Related methods:

- `get_control()` in `Current` interface
- `create_with_name()` in `TransactionFactory` interface
- `recreate()` in `TransactionFactory` interface

```
hash_top_level_tran()
```

```
unsigned long hash_top_level_tran();
```

Most applications will not normally call this method.

Because `VisiTransact` does not support nested transactions, every transaction is a top-level transaction. Therefore, this method is equivalent to the `hash_transaction()` method.

This method returns a hash code for the transaction associated with the target object. Each transaction has a single hash code. The hash code can be used to efficiently compare `Coordinator`s for inequality against the hash codes of other transactions. If the hash codes of two `Coordinator`s are not equal, then they represent different transactions. If two hash codes are equal, then `Coordinator::is_same_transaction()` must be used to guarantee equality or inequality, because two `Coordinator`s might have the same hash code but, in fact, represent two different transactions.

Included in the `Coordinator` interface in **CosTransactions.idl**.

Related methods:

- `hash_transaction()`

For more information, see the *VisiTransact Guide*.

hash_transaction()

```
unsigned long hash_transaction();
```

Most applications will not normally call this method.

This method returns a hash code for the transaction associated with the target object. Each transaction has a single hash code. The hash code can be used to efficiently compare Coordinators for inequality against the hash codes of other transactions. If the hash codes of two Coordinators are not equal, then they represent different transactions. If two hash codes are equal, then `Coordinator::is_same_transaction()` must be used to guarantee equality or inequality, because two Coordinators might have the same hash code but, in fact, represent two different transactions.

Included in the `Coordinator` interface in **CosTransactions.idl**.

The example below shows a method that uses `hash_transaction()` to efficiently reject unequal Coordinators.

```
CORBA::Boolean are_same(Coord1, Coord2)
{
    CORBA::ULong hash1 = Coord1->hash_transaction();
    CORBA::ULong hash2 = Coord2->hash_transaction();
    if(hash1 != hash2)
    {
        return 0;
    }
    else
    {
        return Coord1->is_same_transaction(Coord2);
    }
}
```

Related methods:

- `hash_top_level_tran()`
- `is_same_transaction()`

For more information, see the *VisiTransact Guide*.

is_ancestor_transaction()

```
boolean is_ancestor_transaction(in Coordinator coord);
```

Because VisiTransact does not support nested transactions, this method returns true if, and only if, the target object and the parameter object refer to the same transaction.

Included in the `Coordinator` interface in **CosTransactions.idl**.

The following parameters are used by this method.

Parameter	Description
<code>coord</code>	The Coordinator with which the target Coordinator is compared.

Related methods:

- `is_same_transaction()`

`is_descendant_transaction()`

```
boolean is_descendant_transaction(in Coordinator coord);
```

Because `VisiTransact` does not support nested transactions, this method returns true if, and only if, the target object and the parameter object refer to the same transaction.

Included in the `Coordinator` interface in **CosTransactions.idl**.

The following parameters are used by this method.

Parameter	Description
<code>coord</code>	The Coordinator with which the target Coordinator is compared.

Related methods:

- `is_same_transaction()`

`is_related_transaction()`

```
boolean is_related_transaction(in Coordinator coord);
```

Because `VisiTransact` does not support nested transactions, this method returns true if, and only if, the target object and the parameter object refer to the same transaction.

Included in the `Coordinator` interface in **CosTransactions.idl**.

The following parameters are used by this method.

Parameter	Description
<code>coord</code>	The Coordinator with which the target Coordinator is compared.

Related methods:

- `is_same_transaction()`

`is_same_transaction()`

```
boolean is_same_transaction(in Coordinator coord);
```

This method returns true if, and only if, the target object and the parameter object both refer to the same transaction.

Included in the `Coordinator` interface in **CosTransactions.idl**.

The following parameters are used by this method.

Parameter	Description
<code>coord</code>	The Coordinator with which the target Coordinator is compared.

For more information, see the *VisiTransact Guide*.

`is_top_level_transaction()`

```
boolean is_top_level_transaction(in Coordinator coord);
```

Because VisiTransact does not support nested transactions, this method always returns true.

Included in the `Coordinator` interface in **CosTransactions.idl**.

The following parameters are used by this method.

Parameter	Description
<code>coord</code>	The Coordinator with which the target Coordinator is compared.

For more information, see the *VisiTransact Guide*.

`register_resource()`

```
RecoveryCoordinator register_resource(in Resource resource) raises(Inactive);
```

This method registers the specified Resource as a participant in the transaction associated with the target object. When the transaction is terminated, the Resource will receive requests to prepare, commit, or rollback the updates performed as part of the transaction. For information on Resource methods, see [“Resource interface” on page 181](#).

This method returns a RecoveryCoordinator that can be used by this Resource during recovery.

Included in the `Coordinator` interface in **CosTransactions.idl**.

The following parameters are used by this method.

Parameter	Description
<code>resource</code>	The Resource object to register.

The following exceptions may be raised when calling this method.

Exception	When thrown
<code>CosTransactions::Inactive</code>	This exception is thrown if the transaction has already been prepared.
<code>CORBA::TRANSACTION_ROLLEDBACK</code>	This exception is thrown if the transaction has been marked for rollback.

Related methods:

- `register_resource()` in Current interface

For more information, see [“RecoveryCoordinator interface” on page 180](#) and [“Resource interface” on page 181](#).

`register_synchronization()`

```
void register_synchronization(in Synchronization synch)
raises(Inactive, SynchronizationUnavailable);
```

This method registers the specified Synchronization object so that it will be notified to perform the necessary processing before and after completion of the transaction. Such methods are described in the description of the `Synchronization` interface; see [“Synchronization interface” on page 185](#).

Included in the `Coordinator` interface in **CosTransactions.idl**.

The following parameters are used by this method.

Parameter	Description
<code>synch</code>	The Synchronization object to register.

The following exceptions may be raised when calling this method.

Exception	When thrown
<code>CosTransactions::Inactive</code>	This exception is thrown if the transaction has already been prepared.
<code>CosTransactions::SynchronizationUnavailable</code>	This exception is never raised.
<code>CORBA::TRANSACTION_ROLLEDBACK</code>	This exception is thrown if the transaction has been marked for rollback.

Related methods:

- `register_synchronization()` in `Current` interface

For more information, see “[Resource interface](#)” on page 181, “[Synchronization interface](#)” on page 185, and see “[VisiTransact basics](#)” in the *VisiTransact Guide*.

`register_subtran_aware()`

```
void register_subtran_aware(in SubtransactionAwareResource resource)
raises(Inactive, SubtransactionsUnavailable);
```

Because `VisiTransact` does not support nested transactions, this method always raises `CosTransactions::SubtransactionsUnavailable`.

Included in the `Coordinator` interface in **`CosTransactions.idl`**.

The following parameters are used by this method.

Parameter	Description
<code>resource</code>	The Resource to be registered with the subtransaction.

The following exceptions may be raised when calling this method.

Exception	When thrown
<code>CosTransactions::Inactive</code>	This exception is never thrown.
<code>CosTransactions::SubtransactionsUnavailable</code>	This exception is thrown whenever this method is invoked.

Related methods:

- `register_resource()`
- `rollback_only()`

```
void rollback_only()
raises (Inactive);
```

This method modifies the transaction associated with the `Coordinator` so that `rollback` is the only possible transaction outcome.

Included in the `Coordinator` interface in **`CosTransactions.idl`**.

The following exceptions may be raised when calling this method.

Exception	When thrown
<code>CosTransactions::Inactive</code>	This exception is thrown if the transaction has already been prepared.

Related methods:

- `get_coordinator()` in Control interface
- `rollback_only()` in Current interface

For more information about invoking `rollback_only()`, see the *VisiTransact Guide*.

RecoveryCoordinator interface

When a Resource is registered with the Coordinator, a `RecoveryCoordinator` is returned. The `RecoveryCoordinator` is implicitly associated with a single Resource registration request and can only be used by that Resource. In case recovery is necessary, the Resource can use the `RecoveryCoordinator` during the recovery process.

Also, the Resource can use the `RecoveryCoordinator` if it needs to know the current status of the transaction. For example, the Resource can set its own timeout, and if commit or rollback does not occur within the timeout, the Resource can invoke `replay_completion()` to determine the status of the transaction.

The following example shows the `RecoveryCoordinator` interface in the **CosTransactions.idl** file.

```

:
interface RecoveryCoordinator
{
    Status replay_completion(in Resource resource)
        raises(NotPrepared);
};
:

```

RecoveryCoordinator methods

`replay_completion()`

```

Status replay_completion(Resource resource)
raises(NotPrepared);

```

This method notifies the VisiTransact Transaction Service that the Resource is available. This method is typically used during recovery, and can be used by the Resource to determine the status of the transaction.

Note This method does not initiate completion.

Included in the `RecoveryCoordinator` interface in **CosTransactions.idl**.

The following parameters are used by this method.

Parameter	Description
<code>resource</code>	The Resource for which the recovery is being undertaken.

The following exceptions may be raised when calling this method.

Exception	When thrown
<code>CosTransactions::NotPrepared</code>	This exception is thrown if <code>replay_completion()</code> is called for a Resource that has not yet been prepared.

Related methods:

- `commit()` in Resource interface
- `register_resource()` in Current interface
- `register_resource()` in Coordinator interface
- `rollback()` in Resource interface

For more information on Status values, see “Status value definitions” on page 157 in “Current interface” on page 151.

Resource interface

VisiBroker VisiTransact uses a two-phase commit protocol to complete a top-level transaction with each Resource registered with it—that is, with each Resource that might change during the transaction. The Resource interface defines the methods invoked by the VisiTransact Transaction Service on each Resource. Each object supporting the Resource interface is implicitly associated with a single top-level transaction.

The following example shows the Resource interface in the `CosTransactions.idl` file.

```

:
interface Resource
{
    Vote prepare()
        raises(
            HeuristicMixed,
            HeuristicHazard
        );
    void rollback()
        raises(
            HeuristicCommit,
            HeuristicMixed,
            HeuristicHazard
        );
    void commit()
        raises(
            NotPrepared,
            HeuristicRollback,
            HeuristicMixed,
            HeuristicHazard
        );
    void commit_one_phase()
        raises(
            HeuristicHazard
        );
    void forget();
};
:

```

VisiBroker VisiTransact provides this interface, but you must provide the implementation in your Resource. A typical application does not implement a Resource.

Resource methods

`commit()`

```
void commit()
raises (NotPrepared
        HeuristicRollback
        HeuristicMixed
        HeuristicHazard
);
```

This method attempts to commit all changes associated with the Resource. If a heuristic outcome exception is raised, the Resource must keep the heuristic decision in persistent storage until the `forget()` method is performed so that it can return the same outcome in case `commit()` is invoked again during recovery. Otherwise, the Resource can immediately forget all knowledge of the transaction.

Included in the `Resource` interface in **CosTransactions.idl**.

The following exceptions may be thrown when calling this method.

Exception	When thrown
<code>CosTransactions::NotPrepared</code>	The <code>commit()</code> method was called before the <code>prepare()</code> method was called.
<code>CosTransactions::HeuristicRollback</code>	A heuristic decision was made and all relevant updates have been rolled back.
<code>CosTransactions::HeuristicMixed</code>	A heuristic decision was made. Some relevant updates have been committed and others have been rolled back.
<code>CosTransactions::HeuristicHazard</code>	A heuristic decision may have been made, the disposition of all relevant updates is not known. For those updates whose disposition is known, either all have been committed or all have been rolled back. If the known updates are a mixture of commits and rollbacks, then the <code>HeuristicMixed</code> exception is raised.

Related methods:

- `commit_one_phase()`
- `rollback()`

For more information, see the *VisiTransact Guide*.

`commit_one_phase()`

```
void commit_one_phase()
raises (HeuristicHazard);
```

The `commit_one_phase()` method requests the Resource to commit all changes made as part of the transaction. This method is an optimization for use when a transaction has only one participating Resource. The `commit_one_phase()` method can be called on the Resource, instead of first calling `prepare()` and then `commit()` or `rollback()`.

If a heuristic outcome exception is raised, the Resource must keep the heuristic decision in persistent storage until the `forget()` method is performed. This enables the Resource to return the same outcome in case `commit_one_phase()` is performed again during recovery. Otherwise, the Resource immediately forgets all knowledge of the transaction.

Included in the `Resource` interface in **CosTransactions.idl**.

If a failure occurs during `commit_one_phase()`, it is called again when the failure is repaired. Since there is only a single Resource, the `HeuristicHazard` exception is used to report heuristic decisions related to that Resource.

The following exceptions may be thrown when calling this method.

Exception	When thrown
<code>CosTransactions::HeuristicHazard</code>	A heuristic decision may have been made, the disposition of all relevant updates is not known. For those updates whose disposition is known, either all have been committed or all have been rolled back.
<code>CORBA::TRANSACTION_ROLLEDBACK</code>	The <code>commit_one_phase()</code> method cannot commit all changes made as part of the transaction.

Related methods:

- `commit()`
- `forget()`
- `prepare()`
- `rollback()`

For more information, see the *VisiTransact Guide*.

forget()

```
void forget();
```

When VisiBroker VisiTransact receives a heuristic exception, it records the exception. The VisiTransact Transaction Service will ultimately call `forget()` on the Resource. This means that the Resource can discard all information about the transaction that raised the heuristic exception. This method is called only if a heuristic exception was raised from `rollback()`, `commit()`, or `commit_one_phase()`.

Included in the `Resource` interface in **CosTransactions.idl**.

Related methods:

- `commit()`
- `commit_one_phase()`
- `rollback()`

For more information, see the *VisiTransact Guide*.

prepare()

```
Vote prepare()
raises(HeuristicMixed
       HeuristicHazard
);
```

This method performs the prepare operation—the first step in the two-phase commit protocol for a Resource object. When finished, the method returns one of these Vote values.

- `VoteReadOnly`—No persistent data associated with the Resource has been modified by the transaction.
- `VoteCommit`—The following data has been saved to persistent storage:
 - 1 All data changed as part of the transaction
 - 2 A reference to the `RecoveryCoordinator` object
 - 3 An indication that the Resource has been prepared
- `VoteRollback`—Some circumstance has caused the Resource to call for a rollback, such as inability to save the relevant data, inconsistent outcomes, or no knowledge of the transaction (which might happen after a crash).

After returning `VoteReadOnly` or `VoteRollback`, the Resource can forget all knowledge of the transaction.

If a heuristic outcome exception is raised, the Resource must save the heuristic decision in persistent storage until the `forget()` method is called so that it can return the same outcome in case `prepare()` is called again.

Included in the `Resource` interface in **CosTransactions.idl**.

The following exceptions may be thrown when calling this method.

Exception	When thrown
<code>CosTransactions::HeuristicMixed</code>	A heuristic decision has been made. Some relevant updates have been committed and others have been rolled back.
<code>CosTransactions::HeuristicHazard</code>	A heuristic decision may have been made, the disposition of all relevant updates is not known. For those updates whose disposition is known, either all have been committed or all have been rolled back. If the known updates are a mixture of commits and rollbacks, then the <code>HeuristicMixed</code> exception is raised.

Related methods:

- `commit_one_phase()`
- `register_resource()` in `Current` interface
- `register_resource()` in `Coordinator` interface

For more information, see the *VisiTransact Guide*.

`rollback()`

```
void rollback()
raises(HeuristicCommit
       HeuristicMixed
       HeuristicHazard
);
```

This method rolls back all updates associated with the Resource object.

If a heuristic outcome exception is raised, the Resource must save the heuristic decision in persistent storage until the `forget()` method is invoked. This enables the Resource to return the same outcome in case `rollback()` is called again during recovery. Otherwise, the Resource immediately forgets all knowledge of the transaction.

Included in the `Resource` interface in **CosTransactions.idl**.

The following exceptions that may be raised when calling this method.

Exception	When thrown
<code>CosTransactions::HeuristicCommit</code>	A heuristic decision was made and all relevant updates have been committed.
<code>CosTransactions::HeuristicMixed</code>	A heuristic decision was made and some relevant updates have been committed, and others have been rolled back.
<code>CosTransactions::HeuristicHazard</code>	A heuristic decision may have been made, the disposition of all relevant updates is not known. For those updates whose disposition is known, either all have been committed or all have been rolled back. If the known updates are a mixture of commits and rollbacks, then the <code>HeuristicMixed</code> exception is raised.

Related methods:

- `commit()`
- `commit_one_phase()`
- `forget()`

For more information, see the *VisiTransact Guide*.

Synchronization interface

The `Synchronization` interface defines methods that enable a transactional object to be notified before the start of the two and one-phase commit protocol, and after its completion, as described in the *VisiTransact Guide*.

In the `CosTransactions` module, the `Synchronization` interface provides two methods:

- `before_completion()`—Ensures that `before_completion()` is invoked before starting to commit a transaction.
- `after_completion()`—Ensures a transactional object is notified after the transaction has been completed. This applies to all transactions whether they were committed or rolled back.

Here are two limitations you should be aware of:

- If the VisiTransact Transaction Service cannot contact your `Synchronization` object while trying to call `before_completion()`, then the transaction will be rolled back. If a `Synchronization` object is unavailable after completion, it will be ignored.
- When the VisiTransact Transaction Service instance recovers from a failure, it does not remember `Synchronization` objects, and will only replay completion and not `Synchronization` objects. If a failure occurs, the `Synchronization` object will not be notified of how the transaction was completed by the VisiTransact Transaction Service.

Note In certain cases, `after_completion()` is called when `before_completion()` was not called. `before_completion()` is called only if a transaction is still continuing towards a commit at the outset of completion. `after_completion()` is always called (unless the VisiTransact Transaction Service crashes before the transaction completes).

`Synchronization` objects are not recoverable. If an instance of a VisiTransact Transaction Service fails, any transactions that are completed will not involve `Synchronization` objects.

Note Although the signatures of these methods are fixed by the `Synchronization` interface, their implementations are user-defined. This enables an application to do custom processing at key points in a transaction—before and after transaction completion.

The following example shows the `CosTransactions` IDL for the `Synchronization` interface.

```

:
interface Synchronization : TransactionalObject
{
    void before_completion();
    void after_completion(in Status status);
};
:

```

Synchronization methods

`after_completion()`

```
void after_completion(in Status status);
```

This is a method that you write that performs customized processing after the completion of the transaction. It is essentially a callback.

Note The `after_completion()` method is always invoked during normal processing.

As shown above, IDL for the `Synchronization` interface inherits from the `TransactionalObject` interface. As a programmer, you are responsible for writing the implementation of an `after_completion()` method that conforms to the IDL.

If `after_completion()` is to be called in processing a particular transaction, the following actions must be taken:

- A Synchronization object must be created—by the transaction originator or some other transaction participant.
- The Synchronization object must be registered—by getting the transaction's Coordinator, and calling the `register_synchronization()` method in Coordinator and Current. See `register_synchronization()` in Coordinator interface. Registration must be done after the transaction is created and before the start of the two-phase commit.

Multiple Synchronization objects can be created and registered for a single transaction.

The VisiTransact Transaction Service calls this method after the two-phase commit protocol completes. As an example of its use, `after_completion()` can be used by a transactional object to discover the outcome of the transaction. This is particularly useful for transactional objects that are not also recoverable objects, and so are not automatically notified of the outcome.

You can call `get_status()` to see whether or not the transaction has been marked for rollback.

Notice that because `Synchronization` inherits from `TransactionalObject`, the transaction context will be available through the Current object.

Included in the `Synchronization` interface in **CosTransactions.idl**.

The following parameters are used by this method.

Parameter	Description
<code>status</code>	A <code>Status</code> value passed by the Terminator to the Synchronization object once the outcome of the transaction has been determined. See “Status value definitions” on page 157 in “Current interface” on page 151 for a list of possible <code>Status</code> values.

All exceptions will be ignored.

Related methods:

- `before_completion()`
- `get_status()` in Current interface
- `commit()` in Terminator interface
- `register_synchronization()` in Current interface
- `register_synchronization()` in Coordinator interface
- `rollback_only()` in Current interface
- `rollback_only()` in Coordinator interface

For more information see the *VisiTransact Guide*.

```
before_completion()
```

```
void before_completion();
```

This is a method that you write to perform customized processing at the onset of the completion of a transaction. It is called only if the transaction is still continuing towards successful completion. It is essentially a callback.

Note The `before_completion()` method is invoked after the application invokes `commit()`, but before the VisiTransact Transaction Service begins transaction completion. The `before_completion()` method is not invoked for a rollback request.

As shown in the beginning of this section the IDL for the `Synchronization` interface inherits from the `TransactionalObject` interface. As a programmer, you are responsible for writing the implementation of a `before_completion()` method that conforms to the IDL.

If `before_completion()` is to be called when processing a particular transaction, the `Synchronization` object must be registered using the `register_synchronization()` method in the `Coordinator` interface. Register the `Synchronization` object from your transactional object or recoverable server. See `register_synchronization()` in `Coordinator` interface. Registration must be done after the transaction is created and before the start of the two-phase commit.

Multiple `Synchronization` objects can be created and registered for a single transaction.

The VisiTransact Transaction Service calls this method after the transaction work has been done but before the two-phase commit protocol starts; that is, before `prepare()` is called on the participating Resource. VisiBroker VisiTransact calls `before_completion()` only if a transaction is still continuing towards a commit at the outset of completion. This means that `Terminator->commit()` was called and the transaction has not been marked for rollback. If `Terminator->rollback()` was called, or the first of several `Synchronization` objects marked the transaction for rollback, or the transaction was already marked for rollback, `before_completion()` calls will not be called again for this transaction.

Within this method, you can ensure the transaction will be rolled back by calling the `rollback_only()` method. You can also call `get_status()` to see whether or not the transaction has been marked for rollback. At the time the method is called, however, you cannot rely upon the status to indicate whether or not the transaction will actually be committed.

Notice that because the `Synchronization` interface inherits from `TransactionalObject`, the transaction context will be available through the `Current` object. This means that `before_completion()` can use all objects on the `Current` object, such as `get_status()` and `get_control()`.

Included in the `Synchronization` interface in **CosTransactions.idl**.

All CORBA exceptions raised by your `Synchronization` objects will result in the transaction being rolled back.

Related methods:

- `after_completion()`
- `get_status()` in `Current` interface
- `commit()` in `Terminator` interface
- `register_synchronization()` in `Current` interface
- `register_synchronization()` in `Coordinator` interface
- `rollback_only()` in `Current` interface
- `rollback_only()` in `Coordinator` interface

For more information see the *VisiTransact Guide*.

VISTransactionService class

The `VISTransactionService` class is provided to help you link an instance of the VisiTransact Transaction Service with your application process.

Its methods, in the `visits.h` file, include:

- `init()`
- `terminate()`

The following section documents these methods.

VISTransactionService methods

`init()`

```
static void init(int &argc, char* const* argv);
```

- W** This method initializes all the instances of the VisiTransact Transaction Service that are linked in your application process. It must be invoked to activate an instance of the VisiTransact Transaction Service that you have linked into your process by adding the `ots_r` library and the `otsinit` object file to the link line.

If you want to initialize the embedded instance of the VisiTransact Transaction Service, the `init()` method must be called. After `ORB_init()` has been invoked, all the recognized VisiTransact arguments will be stripped from the original parameter list so that they will not interfere with any other argument processing that your client program requires.

- Caution** If the in-process VisiTransact Transaction Service instance has been de-activated using `terminate()`, `vshutdown`, or the Console, do not invoke `init()` again.

Included in the `VISTransactionService` interface in `visits.h`.

The following parameters are used by this method.

Parameter	Description
<code>argc</code>	The number of arguments being passed to the <code>init()</code> method.
<code>argv</code>	The actual arguments being passed to the <code>init()</code> method.

- Note** The `argc` and `argv` parameters should be `argc` and `argv` from the `main` function. For more information, see the *VisiTransact Guide*.

`terminate()`

```
static void terminate();
```

- W** This method will cleanup all the instances of the VisiTransact Transaction Service that have been initialized by a call to `init()`. This method will not bring down your application process unless the command line option `OTSexit_on_shutdown` is set to 1.

If this option is either not set, or set to 0, it will deactivate the VisiTransact Transaction Service objects registered with the Smart Agent but will NOT bring down your application process.

Included in the `VISTransactionService` interface in `visits.h`.

For more information, see the *VisiTransact Guide*.

VISSessionManager module

This section introduces the `VISSessionManager` module and describes its classes, data types, structures and methods.

Looking at the module

The Session Manager is a component that allows an application to obtain pre-configured database connections. The Session Manager insulates applications from the database-specific requirements for connection handles and thread management. Once a connection is obtained using the Session Manager, the transaction is coordinated automatically by the VisiTransact Transaction Service. The application developer is free from creating code to incorporate the database's participation in the transaction—the application code only needs to address issues concerning the data it requires from the database.

The Session Manager and its associated Resources provide complete transactional access to the DBMS. Full two-phase commit capability is supported by the XA implementation of the Session Manager along with its Resource implementation (the XA Resource Director). For distributed transactions, the Session Manager, in conjunction with the VisiTransact Transaction Service, performs the XA interface calls to include the application's work on that database.

Alternatively, the `DirectConnect` version of the Session Manager provides optimized transactional access using an integrated Resource, but requires a more restrictive programming model.

Your applications can use the following interfaces from the Session Manager module:

- `Connection`—Represents a database connection with transaction support.
- `ConnectionPool`—A factory interface that allocates connections to clients.

These interfaces are pseudo IDL, not true IDL, because the `Connection` and `ConnectionPool` objects must be available locally in the process. Access to the `ConnectionPool` is obtained using the `resolve_initial_references()` call.

Currently, only C++ Session Manager interfaces are available.

The following code sample is the IDL for the `VISSessionManager` module.

```
#ifndef _vissessionmanager_idl_
#define _vissessionmanager_idl_
#include <CosTransactions.idl>
#pragma prefix "visigenic.com"

module VISSessionManager
{
    struct Attribute
    {
        string name;
        string value;
    };
    typedef sequence<Attribute> Attributes;

    struct ErrorInfo
    {
        string reason;
        string subsystem;
        unsigned long code;
    };
    typedef sequence<ErrorInfo> ErrorInfos;
}
```

```

exception Error
{
    ErrorInfos info;
};

interface Connection
{
    enum ReleaseType
    {
        MarkSuccess,
        MarkForRollback
    };

    typedef unsigned long long NativeConnectionHandle;

    NativeConnectionHandle getNativeConnectionHandle()
        raises(Error);

    Attributes getAttributes()
        raises(Error);

    string getInfo(in string info_type)
        raises(Error);

    boolean isSupported(in string support_type)
        raises(Error);

    void hold(in unsigned long timeout)
        raises(Error);

    void resume()
        raises(Error);

    void release(in ReleaseType type)
        raises(Error);
    void releaseAndDisconnect()
        raises(Error);
}; // Connection

interface ConnectionPool
{
    exception ProfileError
    {
        string reason;
        unsigned long code;
    };

    Connection getConnection(in string profile_name)
        raises(ProfileError, Error);

    Connection getConnectionWithCoordinator(in string profile_name,
        in CosTransactions::Coordinator coord)
        raises(ProfileError, Error);

    Attributes getProfileAttributes(in string profile_name)
        raises(ProfileError);

}; // interface ConnectionPool
}; // module VISessionManager

#pragma prefix ""
#endif // _vissessionmanager_idl_

```

Structures

The `VISSessionManager` module defines as data types the following structures:

- `ErrorInfo`—This structure is used in exceptions.
- `Attribute`—This structure describes a connection attribute. Attribute values are always represented as strings in the `Attribute` structure.

The following code sample shows the `ErrorInfo` structure in the `VISSessionManager` module.

```
struct ErrorInfo
{
    string reason;
    string subsystem;
    unsigned long code;
};
```

The following table defines the members of the `ErrorInfo` structure.

Table 10.4 Members of the `ErrorInfo` structure

Member	Description
<code>reason</code>	A string that contains an explanation of the error.
<code>subsystem</code>	The basic component which generated the error.
<code>code</code>	An error code number.

Currently, the subsystems that may generate errors for the Session Manager are:

- "Session Manager"—The basic body of the Session Manager module.
- "XA Native"—The error originated in an XA call made to the database.
- "Oracle"—The error occurred in a direct call to Oracle APIs.

The following code sample shows the `Attribute` structure in the `VISSessionManager` module.

```
struct Attribute
{
    string name;
    string value;
};
```

The following table defines the members of the `Attribute` structure.

Table 10.5 Members of the `Attribute` structure

Member	Description
<code>name</code>	A string that indicates the type of attribute.
<code>value</code>	A value for the named attribute.

The following attributes exist for all the Resource Managers and will be specified in all connection profiles.

Table 10.6 Connection profile attributes

Attribute	Default setting	Description
<code>database_name</code>	None	A character string representing the database name.
<code>userid</code>	None	A character string representing the user identification to be used when getting the connection.
<code>password</code>	None	A character string representing the database's password.

Exceptions

If a Session Manager method is going to get an exception, most likely it will be this one. However, it can also get a standard CORBA exception.

The following code sample shows the Error exception in the VISessionManager module.

```
exception Error
{
    ErrorInfos info;
};
```

ErrorInfos is a sequence of structures (representing an error stack) returned to the application. The application can query the sequence about how many errors are in the sequence and can see in what layer the error occurred. Then it can access the information about an error one at a time. The structure provides the following information:

- Reason for the error
- Subsystem (module) where the error occurred
- Error code for this particular error

The following code sample shows an example of error iterating.

```
:
try
{
    // Ask the pool for a database connection
    // Use the database profile "quickstart"
    conn = _pool->getConnection("quickstart");

    // get a connection handle to use for native OCI calls
    lda_ptr = (Lda_Def*) conn->getNativeConnectionHandle();
}
catch(const VISessionManager::Error& ex)
{
    cerr << "Session Manager error:\n";
    // print out all the error messages
    for(CORBA::ULong i = 0; i < ex.info.length(); i++)
    {
        cerr << " " << ex.info[i]. subsystem
            << "-" << ex.info[i].code
            << ": " << ex.info[i].reason
            << endl;
    }
    throw ApplicationException();
    // This would be something an application would define.
}
:
```


ConnectionPool interface

The `ConnectionPool` interface provides access to the Session Manager's pool of database connections. There is one logical `ConnectionPool` per process, regardless of what type(s) of connections it will manage. The `ConnectionPool` uses a specified connection profile to allocate connections. When the application requests a connection, the `ConnectionPool` object attempts to find one already open in the pool that it can use. If the `ConnectionPool` object can not find an appropriate connection, it will open a new one.

The `ConnectionPool` interface also provides a method that enables the application to find out the configuration profile's attributes without allocating a connection to the database. The application can query from the connection pool using `getProfileAttributes()` and the profile's attributes will be returned.

The C++ header files that you include must also correspond to your choice of interfaces. `VISessionManager_c.hh` is the client header file generated from `VISessionManager.idl`.

Note Do not generate your own client stub header file. You must use the supplied client header files to ensure compatibility with the Session Manager object libraries.

Obtaining a ConnectionPool object reference

The following steps describe the general process for obtaining a reference to a `ConnectionPool` object, and are followed by a code example.

- 1 Call the ORB `resolve_initial_references()` method, passing the object type `VISessionManager::ConnectionPool`.
- 2 Narrow the returned object to a `VISessionManager::ConnectionPool`.

The following code sample is an example of obtaining a `ConnectionPool` object reference in C++.

```

:
{
CORBA::ORB_var orb = CORBA::ORB_init();
CORBA::Object_var object =
  orb->resolve_initial_references("VISessionManager::ConnectionPool");
VISessionManager::ConnectionPool_var pool =
  VISessionManager::ConnectionPool::_narrow(object);
:
}

```

Using ConnectionPool object references

The `ConnectionPool` object reference is valid for the entire process under which you create it; you can use it in any thread. You can either make multiple calls to obtain references to the `ConnectionPool` object or use just one reference throughout the entire process, saving the overhead of numerous `resolve_initial_references()` calls.

Exceptions

When an error occurs having to do with the profile, an operation on `ConnectionPool` may throw a `ConnectionPool::ProfileError` exception.

The following code sample shows the `ProfileError` exception in the `ConnectionPool` interface.

```

exception ProfileError
{
  string reason;
  unsigned long code
};

```

For more information about `Error`, see [“Exceptions” on page 192](#).

Methods

The methods in the `ConnectionPool` interface include:

- `getConnection()`
- `getConnectionWithCoordinator()`
- `getProfileAttributes()`

The following sections document these methods.

`getConnection()`

This VisiBroker-only method allocates a database connection, transparently associates the connection with a transaction, and returns a `Connection` object to the application.

Before a thread can call `getConnection()`, it must have an active transaction context. The method `getConnectionWithCoordinator()` may be used to get connections for an explicitly-specified transaction.

If this method raises an `Error` exception, see “Error codes” in the *VisiTransact Guide* for information.

Interface

`ConnectionPool` in **VISessionManager.idl**

Signature

```
Connection getConnection(in string profile_name)
raises(VISessionManager::ConnectionPool::ProfileError,
      VISessionManager::Error);
```

Parameters

The following parameters are used by this method.

Parameter	Description
<code>profile_name</code>	The name of the profile which describes the attributes for the requested connection.

Example

The following code sample is an example of using the `getConnection()` method.

```
:
VISessionManager::ConnectionPool_var pool;
// Ask the pool for a database connection
VISessionManager::Connection_var conn = pool->getConnection("quickstart");
:
```

For more information, see “Data access using the Session Manager” in the *VisiTransact Guide* for information about connection pooling.

See also

- “[ConnectionPool interface](#)” on page 193

getConnectionWithCoordinator()

This VisiBroker-only method allocates a connection using an explicitly-specified transaction Coordinator, and returns a Connection object to the application. The connection is transparently associated with the transaction. If this method raises an `Error` exception, see “Error codes” in the *VisiTransact Guide* for information.

To allocate a connection for an implicit transaction context, use `getConnection()`.

Interface

ConnectionPool in **VISSessionManager.idl**

Signature

```
Connection getConnectionWithCoordinator(in string profile_name,
                                       in CosTransactions::Coordinator coord)
raises(VISSessionManager::ConnectionPool::ProfileError,
      VISSessionManager::Error);
```

Parameters

The following parameters are used by this method.

Parameter	Description
profile_name	The name of the connection for which a connection will be obtained.
coord	The Coordinator object reference representing the transaction which should be associated with this connection.

Example

The following code sample is an example of using the `getConnectionWithCoordinator()` method.

```
:
VISSessionManager::ConnectionPool_var pool;
// Ask the pool for a database connection
// Use the database profile "quickstart"
conn = pool->getConnectionWithCoordinator("quickstart", coordinator);
:
```

For more information, see “Data access using the Session Manager” in the *VisiTransact Guide* for information about connection pooling.

getProfileAttributes()

This VisiBroker-only method may be used to query attributes in a profile without allocating a connection. The application passes in the name of a connection profile with this method and the ConnectionPool object returns the connection profile's attributes to the application.

Note You can also view attributes of a connection that is currently open, using `getAttributes()`.

The Session Manager returns a reference to an Attributes object in response to this method call. Applications should use an object of the following type to hold this object and ensure proper memory management:

```
VISSessionManager::Attributes_var
```

Interface

ConnectionPool in **VISSessionManager.idl**

Signature

```
Attributes getProfileAttributes(in string profile_name)
raises (VISessionManager::ConnectionPool::ProfileError);
```

For a listing of connection profile attributes, see the table [“Connection profile attributes” on page 191](#).

Parameters

The following parameters are used by this interface.

Parameter	Description
<code>profile_name</code>	The name of the connection profile whose attribute values are returned.

Example

The following code sample is an example of using `getProfileAttributes()`.

```
:
VISessionManager::ConnectionPool pool;
VISessionManager::Attributes_var attrs;
attrs = pool->getProfileAttributes("quickstart");
CORBA::ULong len = attrs->length();
for (CORBA::ULong i=0; i<len; i++)
{
    cout << "Attribute " << i << ": " << attrs[i].name
        << " = " << attrs[i].value << endl;
}
:
```

For more information, see [“Data access using the Session Manager” in the *VisiTransact Guide*](#) and [“ConnectionPool interface” on page 193](#).

Connection interface

The `Connection` interface gives the application access to transaction-configured database connections.

Instead of creating a `Connection` object directly, you obtain a reference to a `Connection` object from the `ConnectionPool` using `getConnection()` or `getConnectionWithCoordinator()`. If the `ConnectionPool` returns one of these objects, it has allocated the database connection and associated it with a transaction.

Once you have invoked its `release()` or `releaseAndDisconnect()` method, any subsequent operations on that connection will result in an exception.

The `Connection` interface offers a method to obtain your native database handle, as well as several methods that allow you to programmatically view information about the connection.

The C++ header files that you include must also correspond to your choice of interfaces. `VISessionManager_c.hh` is the client header file generated from `VISessionManager.idl`.

Data types

The `Connection` interface defines the data type `ReleaseType`. For more information about `ReleaseType`, see [“release\(\)” on page 200](#).

Methods

The methods in the `Connection` interface include:

- `getAttributes()`
- `getInfo()`
- `getNativeConnectionHandle()`
- `hold()`
- `isSupported()`
- `release()`
- `releaseAndDisconnect()`
- `resume()`

The following sections document these methods.

`getAttributes()`

This VisiBroker-only method is used to return the values of configuration profile attributes for a connection that is currently allocated. The Session Manager allocates an `Attributes` object and returns a reference to that object. Applications should use an object of the following type to hold this object and ensure proper memory management.

```
VISessionManager::Attributes_var
```

Interface

`Connection` in `VISessionManager.idl`

Signature

```
Attributes getAttributes()
raises (VISessionManager::Error);
```

For a listing of connection profile attributes, see the table [“Connection profile attributes” on page 191](#).

Parameters

None.

Example

The following code sample is an example of using `getAttributes()`.

```
:
VISessionManager::Connection_var conn;
VISessionManager::Attributes_var attrs;
attrs = conn->getAttributes();
CORBA::ULong len = attrs->length();
for (CORBA::ULong i=0; i<len; i++)
{
    cout << "Attribute " << i << ": " << attrs[i].name
        << " = " << attrs[i].value << endl;
}
:
```

For more information, see “Data access using the Session Manager” in the *VisiTransact Guide*.

`getInfo()`

This exclusive VisiBroker method is used to query the characteristics of a Session Manager implementation for a particular Resource Manager. For example, an application can query for the version of the Session Manager. Applications should use an object of the following type to hold the returned value and ensure proper memory management.

```
CORBA::String_var
```

InterfaceConnection in **VISessionManager.idl****Signature**

```
string getInfo(in string info_type)
raises (VISessionManager::Error);
```

Parameters

The following parameters are used by this method.

Parameter	Description
info_type	The information type to be returned.

A list of information types for a particular type of Session Manager, if any, can be found in the VisiBroker Integrated Transaction Service *Data Access Guide*.

These information types are available for all types of Session Managers:

- "version"—Returns the version number of the generic Session Manager. The version number is returned in a 5-field string which is standard in the VisiBroker utility `vbver`. This information is to be used for informational purposes.
- "version_rm"—Returns the version number of the Resource Manager-specific component of the Session Manager. This information is to be used for informational purposes.

The following code sample is an example of using `getInfo()`.

```
:
VISessionManager::Connection_var conn;
CORBA::String_var info = conn->getInfo("version");
:
```

For more information see “Data access using the Session Manager” in the *VisiTransact Guide*.

getNativeConnectionHandle()

This VisiBroker-exclusive method is used to return the native connection handle to a Resource Manager (generally, a database), for the database connection represented by the Connection object. Applications can use the native handle to make calls to the Resource Manager's native API. Applications should not disconnect using Resource Manager API calls on this handle but instead use the `release()` or `releaseAndDisconnect()` methods to release the connection.

This is a low overhead operation that can be invoked as often as necessary to obtain the native connection handle.

Note This method does not allocate connections. See “[getConnection\(\)](#)” on page 194 or “[getConnectionWithCoordinator\(\)](#)” on page 195 for more information.

InterfaceConnection in **VISessionManager.idl****Signature**

```
NativeConnectionHandle getNativeConnectionHandle()
raises (VISessionManager::Error);
```

Parameters

None.

Example

The following code sample is an example of using the `getNativeConnectionHandle()` method.

```

:
VISSessionManager::Connection_var conn;
//get a connection handle
lda = (Lda_Def *) conn->getNativeConnectionHandle();
:

```

For more information, see “Data access using the Session Manager” in the *VisiTransact Guide* for information about connection pooling.

hold()

Caution Using `hold()` monopolizes the connection and affects performance. Use `hold()` only when it is necessary.

This `VisiBroker`-exclusive method notifies the Session Manager that the thread of control is leaving the current process and intends to return. The application guarantees it will not use this connection handle until `resume()` has been invoked.

The Session Manager requires that it be notified if no thread in the current process is active with respect to this connection. The main reason for this requirement is that if the requester fails or is otherwise unable to return to this process to release its Resources, the Session Manager must be able to clean up any Resources used for this connection including database locks and resources. If the Session Manager does not have knowledge of whether or not the application is still actively using the connection, it cannot dissociate the transaction and proceed with cleanup.

The `timeout` parameter specifies the time in seconds that the Session Manager should wait before timing out the connection and cleaning up its Resources. As part of the cleanup process, the connection is returned to the `ConnectionPool` and the transaction is marked for rollback.

Your application can send multiple `hold()` requests with no intervening `resume()` calls. If `hold()` is called twice, the timer is reset with the new value at each call. For example, if you send `hold(60)` at 8:42:30, it would expire at 8:43:30. However, if you subsequently invoke `hold(45)` at 8:42:50, the timer would expire at 8:43:35 because it had been reset by the second `hold()` call.

Note Some database Session Manager implementations may not support this method. Your application can use `isSupported()` to query whether the Session Manager supports the `hold()` method or not. You can find more information about this in the *VisiBroker Integrated Transaction Service Data Access Guide*.

Before the `Connection` object or the corresponding database connection handle can be used again, `resume()` must be called on the `Connection` object.

Interface

`Connection` in `VISSessionManager.idl`

Signature

```

void hold(in unsigned long timeout)
raises(VISSessionManager::Error);

```

Parameters

The following parameters are used by this method.

Parameter	Description
<code>timeout</code>	The time in seconds for which the Session Manager should wait before releasing the connection back to the pool and marking the transaction for rollback. If you pass in zero, an exception is raised.

Example

```

:
VISessionManager::Connection_var conn;
conn->hold(60);
:

```

For more information, see “Data access using the Session Manager” in the *VisiTransact Guide* and [“ConnectionPool interface” on page 193](#).

isSupported()

This VisiBroker-exclusive method is used to query the supported types of a Session Manager implementation for a particular Resource Manager. For example, an application can query whether the Session Manager supports the `hold()` method or not.

Interface

Connection in **VISessionManager.idl**

Signature

```

boolean isSupported(in string support_type)
raises (VISessionManager::Error);

```

Parameters

The following parameters are used by this method.

Parameter	Description
<code>support_type</code>	The support type to be returned.

The following support types are available for all types of Session Managers.

- `"hold"`—Returns true if the `hold()` method is supported; otherwise, returns false.
- `"thread_portable"`—Returns true if the connections may be used in other threads than the one that made the connection; otherwise, returns false.

```

:
VISessionManager::Connection_var conn;
CORBA::Boolean isPortable = conn->isSupported("thread-portable");
:

```

For more information, see “Data access using the Session Manager” in the *VisiTransact Guide* and [“ConnectionPool interface” on page 193](#).

release()

This VisiBroker-exclusive method releases a connection back to the ConnectionPool and marks the transaction for commit or rollback. If you do not call this method, the transaction is marked for rollback when the Connection object destructs.

When the application invokes `release()`, the state in the Connection is cleaned out. Any further calls on that Connection will raise `CORBA::BAD_OPERATION`.

You can reacquire the connection later to perform further work on the same transaction.

Note The `release()` call does not release the Connection object in the sense of the `CORBA_release()` method. It indicates to the ConnectionPool that the underlying database connection will no longer be needed by the application. The application will still need to de-allocate the Connection object. The easiest way to accomplish this is to hold the Connection object in a `Connection_var` object.

Interface

Connection in **VISessionManager.idl**

Signature

```
void release(in ReleaseType type)
raises (VISSessionManager::Error);
```

ReleaseType

The definition for the `ReleaseType` is:

```
enum ReleaseType
{
    MarkSuccess,
    MarkForRollback
};
```

The descriptions for the `ReleaseType` values are shown in the Parameters table.

Parameters

The following parameters are used by this interface for `release()`, and values for the `ReleaseType`.

Parameter	Description
<code>type</code>	The flag to indicate whether this portion of the transaction was successful or not. If <code>MarkSuccess</code> is passed then the transaction proceeds normally. Otherwise, if <code>MarkForRollback</code> is passed then the transaction is marked for rollback.
<code>MarkSuccess</code>	This portion of the transaction was successful. The transaction proceeds normally.
<code>MarkForRollback</code>	The transaction is marked for rollback. Since the rollback must come from the VisiTransact Transaction Service, there may be some delay between calling <code>release(MarkForRollback)</code> and the work actually rolling back.

Example

The following code sample is an example of using the `release()` method.

```
:
VISSessionManager::Connection_var conn;
conn->release(VISSessionManager::Connection::MarkSuccess);
:
```

For more information, see “Data access using the Session Manager” in the *VisiTransact Guide*.

releaseAndDisconnect()

This VisiBroker-exclusive method forces the database connection to close completely and marks the transaction for rollback. The connection is not returned to the ConnectionPool for re-use. This method is used if the application detects something wrong with the connection and wants to make sure the connection will not be reused.

When the application invokes `releaseAndDisconnect()`, the state in the connection is cleaned out. Any further calls on that Connection will raise `CORBA::BAD_OPERATION`.

Interface

Connection in **VISSessionManager.idl**

Signature

```
void releaseAndDisconnect()
raises (VISSessionManager::Error);
```

Parameters

None.

Example

The following is an example of using the `releaseConnection()` method.

```

:
VISessionManager::Connection_var conn;
cerr << "Profile error: " << ex.code << ex.reason << endl;
conn->releaseAndDisconnect();
return balance;
:

```

For more information, see “Data access using the Session Manager” in the *VisiTransact Guide* and “[ConnectionPool interface](#)” on page 193.

resume()

This VisiBroker-exclusive method is used after a `hold()` to indicate to the Session Manager that the thread of control for this Connection is now back in process. This cancels the timeout associated with the `hold()` and guarantees that the Session Manager will not modify the underlying connection in any way that would cause conflicts with an active application. Calling `resume()` when the Connection has not been placed in the hold state results in an `Error` exception, but does not modify the transaction or connection state.

Note Between the `hold()` and `resume()` calls, the application is not allowed to make any other calls on the `Connection` object or its associated native database handle. If the `hold()` call timeout expires in this interval, the Session Manager has the right to release the connection and mark the transaction for rollback. This is to ensure that resources held in the application server by that transaction are not left forever if a client dies or never calls again.

Interface

Connection in `VISessionManager.idl`

Signature

```

void resume()
raises(VISessionManager::Error)

```

Parameters

None.

Example

The following is an example of using the `resume()` method.

```

:
VISessionManager::Connection_var conn;
conn->resume();
:

```

For more information, see “Data access using the Session Manager” in the *VisiTransact Guide*.

The ITSDataConnection class

The Pluggable Resource Interface is a component that implements a set of predefined interfaces to allow transactional applications to use databases as their persistent storage in transactions managed by Borland VisiTransact. More information see “Pluggable Database Resource Module for VisiTransact” in the *VisiTransact Guide*.

This class is defined below.

```
class ITSDataConnection
{
public:
virtual void connect() = 0;
virtual void disconnect() = 0;
virtual void rollback() = 0;
virtual void commit() = 0;
virtual xa_switch_t* xa_switch() { return 0; }
virtual const char* xa_open_string() { return 0; }
virtual const char* xa_close_string() { return 0; }
virtual void* native_handle() { return 0; }
};
```

The methods in ITSDataConnection class can be divided into three groups:

- native handle acquisition interface
- local transaction connection and completion interface
- global transaction connection and completion interface

Native handle acquisition interface

```
void* native_handle();
```

This function is used to get access to the native APIs for a database supported by the module. The return value is a void pointer, allowing the implementation to return anything necessary to manipulate data in the database. A transactional application can obtain this pointer through `getNativeConnectionHandle()`, in which the Session Manager Connection Manager will call the `native_handle()` and return the pointer back to the application.

Any pluggable module must implement this function.

Local transaction connection and completion interface

Pluggable modules that support the local transaction must implement these functions.

These four methods is used by Session Manager Connection Manager to inform the database of the start and completion of local transactions.

void connect();

When it is called, it establishes the connection to the database and tells the database that a local transaction begins.

void disconnect();

When it is called, it means the connection, if established, is no longer needed. So the connection can be closed.

void rollback();

It tells the database to commit the transaction.

void commit();

It tells the database to rollback the transaction.

Global transaction connection and completion interface

Pluggable modules that support global transactions must implement the functions.

The session manager uses X-open's XA interface to talk to a XA conformable database.

xa_switch_t* xa_switch();

All the Session Manager Connection Manager need from the pluggable module is a pointer to a xa_switch_t data structure which contains all the XA APIs as defined in the xa.h. The xa_switch() function is just for this purpose. Whenever being called, it must return a valid pointer to this data.

Usually the specific database implements and exposes the xa_switch_t to its clients. The name of that data struct varies from database to database. For example, Oracle9i implements its xa_switch_t as a global variable named xaosw.

This function is also used by Session Manager Connection Manager to judge the type of a connection. If the function returns zero, the session manager will treat the connection as DC type, otherwise it takes the connection as XA type.

Pluggable modules that support global transactions must implement the function and must not return zero.

const char* xa_open_string();

When called, it returns a string used as argument to xa_open() call.

const char* xa_close_string();

When called, it returns a string used as argument to xa_close() call.

The two methods are called by the session manager to get database specific parameters to open or close a XA connection to a database. The returned string from the xa_open_string() call will be used in the call on xa_open() and the returned string from the xa_close_string() is used in xa_close().

Once called for an XA connection, the session manager will keep the returned values for later use. The implementation does not need to keep the validity of the returned pointer all the time.

Chapter 11

Native Messaging Interfaces and Classes

This section describes the interfaces and classes associated with the Native Messaging.

RequestAgent

```
class NativeMessaging::RequestAgent : public virtual CORBA_Object
```

The Request Agent interface defines operations of the Native Messaging Request Agent.

Include File

Include the `NativeMessaging_c.hh` file when you use this class.

IDL definition

```
module NativeMessaging {
  interface RequestAgent {
    exception DuplicatedRequestTag {};
    exception PollingGroupIsEmpty {};
    exception RequestNotExist {};

    Request create_request(
      in RequestDesc desc) raises (DuplicatedRequestTag);
    RequestTagSeq poll(
      in string          polling_group,
      in unsigned long  timeout,
      in boolean        unmask) raises
(PollingGroupIsEmpty);
    void destroy_request(
      in Request req) raises (RequestNotExist);
  };
};
```

RequestAgent Methods

create_request

```
virtual ::CORBA::Object_ptr create_request(const NativeMessaging::RequestDesc&
    _desc);
```

This method creates and returns an asynchronous method invocation request object in the Request Agent.

Parameter	Description
<code>_desc</code>	The <code>RequestDesc</code> structure containing information about the target object and async request.

The method throws `DuplicatedRequestTag` exception.

poll

```
virtual NativeMessaging::RequestTagSeq* poll(const char* _polling_group,
    ::CORBA::Ulong _timeout,
    ::CORBA::Boolean _umask);
```

The method returns the sequence of request tags whose replies are ready.

Parameter	Description
<code>_polling_group</code>	The name of the polling group
<code>_timeout</code>	The timeout interval in “milliseconds” to wait if the polling group has no readily available replies. The values have following meanings: <ul style="list-style-type: none"> ■ <code>timeout > 0</code> poll will block for that much time. If after the timeout, there are still no replies available, an empty sequence of request tags is returned. ■ <code>timeout=0</code> poll will not block. If there are any replies available, their tags will be returned to the caller. If there are no replies available, an empty sequence is returned. ■ <code>timeout < 0</code> (or <code>timeout=2^(32-1)</code>) poll will block until a reply is available.
<code>_unmask</code>	If this flag is false, subsequent calls to poll on the same polling group will also return the request tags returned in the previous polls, until those request get destroyed either as a result of manual or automatic trash. If this flag is true, once a request tag is returned in the poll, it will not appear in subsequent polls.

This method throws `PollingGroupIsEmpty` exception.

destroy_request

```
virtual void destroy_request(::CORBA::Object_ptr _req);
```

This method destroys an async request.

Parameter	Description
<code>_req</code>	The async request object reference to be destroyed.

This method throws `RequestNotExist` exception.

RequestDesc

```
struct NativeMessaging::RequestDesc;
```

A descriptor structure containing all the information needed to service a async request.

Include File

Include the `NativeMessaging_c.hh` file when you use this struct.

IDL Definition

```
module NativeMessaging {
    typedef Object          Request;
    typedef sequence<octet> OctetSeq;
    typedef OctetSeq       RequestTag;
    typedef sequence<RequestTag> RequestTagSeq;
    typedef OctetSeq       Cookie;

    struct RequestDesc {
        Object          target;
        string          repository_id;
        ReplyRecipient reply_recipient;
        Cookie          the_cookie;
        string          polling_group;
        RequestTag      request_tag;
        PropertySeq     properties;
    };
};
```

RequestDesc Fields

Field	Description
target	Reference of the target object, on which client wish to invoke an operation. CORBA::BAD_PARAM exception will result if a null value is passed.
repository_id	Repository id of the target object. If this is an empty string, request agent will try to extract the rep id from the target object reference. If rep id empty here and also null or empty in the target IOR reference, a CORBA::BAD_PARAM exception is thrown. Clients can also use a repository id of *. This acts as a wild card and is_a operation on the request object returns true for any repository id.
reply_recipient	The reference of the reply recipient (or reply handler) when using callback model. If this reference is not null then the Request Agent will call its <code>reply_available</code> method when a reply is ready.
the_cookie	A user specified sequence of octets. It will be sent to the reply_recipient when <code>reply_available</code> is called. The information inside the cookie is user defined.
polling_group	A user assigned polling group name. The group name is scoped inside the Request Agent. Group names are not uniquely used. If a non-empty group name string is assigned and the Request Agent doesn't have a polling group with the same name, a new group with that name will be implicitly created. However, if a group already exists, the created request object is inserted into that group.

Field	Description
request_tag	User assigned. If non-empty, it uniquely identifies the request in the group. If another request in the group has the same tag, create_request method throws <code>DuplicatedRequestTag</code> exception.
properties	Sequence of Property structure. Currently only one Property value is defined (see Property structure)

ReplyRecipient

```
class NativeMessaging::ReplyRecipient : public virtual CORBA_Object
```

Defines the interface for callback reply recipient.

Include File

Include the `NativeMessaging_c.hh` file when you use this class.

```
module NativeMessaging {
interface ReplyRecipient {
    void reply_available(
        in Request reply_holder,
        in string operation,
        in Cookie the_Cookie);
};
};
```

ReplyRecipient methods

reply_available

```
virtual void reply_available(::CORBA::Object_ptr _reply_holder, const char*
_operation, const NativeMessaging::OctetSeq& _the_Cookie);
```

Parameter	Description
_reply_holder	async request for which reply is received.
_operation	operation invoked by the client.
_the_Cookie	cookie passed by the client when creating the request.

REPLY_NOT_AVAILABLE

This constant defines the `CORBA::NO_RESPONSE` exception minor code value thrown by the `RequestAgent` to the polling client when the reply for a request is not available.

Include File

Include the `NativeMessaging_c.hh` file when you use this constant.

IDL definition

```
module NativeMessaging {
    const unsigned long REPLY_NOT_AVAILABLE = 100;
};
```

Property

```
struct NativeMessaging::Property;
```

Holds a symbolic property `name` and its `value` inside an `CORBA::Any`.

Include File

Include the `NativeMessaging_c.hh` file when you use this struct.

IDL definition

```
module NativeMessaging {
    struct Property {
        string name;
        any value;
    };
};
```

Property Fields

Field	Description
<code>name</code>	The name of the property. Currently only one name is recognized: <code>RequestManualTrash</code>
<code>value</code>	The value of the property. The <code>RequestManualTrash</code> has a value of type <code>boolean</code> : If set to <code>true</code> , the request is destroyed manually by calling <code>destroy_request</code> method. If set to <code>false</code> , the request is destroyed automatically once the reply is read (default).

PropertySeq

```
class NativeMessaging::PropertySeq : private VISResource
```

A Sequence of `Property` that is passed inside `RequestDesc` while creating `async Request`.

Include File

Include the `NativeMessaging_c.hh` file when you use this class.

OctetSeq

```
class NativeMessaging::OctetSeq : private VISResource
```

This class represents a sequence of octets. Similar to `CORBA::OctetSeq` but defined here to make the `NativeMessaging.idl` independent of any other IDL.

Include File

Include the `NativeMessaging_c.hh` file when you use this class.

RequestTag

```
typedef OctetSeq RequestTag;
```

An octet sequence identifying a request inside a polling group.

Include File

Include the `NativeMessaging_c.hh` file when you use this class.

RequestTagSeq

```
class NativeMessaging::RequestTagSeq : private VISResource
```

Instances of this class are returned by the `RequestAgent`'s `poll` method when group polling is performed. Each element in the sequence is a `RequestTag`; the octet sequence identifying a request inside the polling group.

Include File

Include the `NativeMessaging_c.hh` file when you use this class.

Cookie

```
typedef OctetSeq Cookie
```

An octet sequence that is passed inside `RequestDesc` while creating async `Request`. The contents inside the `Cookie` are user defined. The `Request Agent` passes this `Cookie` to `ReplyRecipient`'s `reply_available` method when callback occurs.

Include File

Include the `NativeMessaging_c.hh` file when you use this type.

DuplicatedRequestTag

```
class DuplicatedRequestTag : public CORBA_UserException
```

This class defines a `UserException` that is raised if the `async` request is created with a polling group name specified and there is another request in the polling group with the same request tag.

Include File

Include the `NativeMessaging_c.hh` file when you use this class.

PollingGroupsEmpty

```
class PollingGroupIsEmpty : public CORBA_UserException
```

This class defines a `UserException` that is raised if `poll` method is called on the `RequestAgent` and:

- There is no group with the specified name.
- The polling group exists but contains no requests that are waiting for replies.

Include File

Include the `NativeMessaging_c.hh` file when you use this class.

RequestNotExist

```
class RequestNotExist : public CORBA_UserException
```

This class defines a `UserException` that is raised if the `destroy_request` method is called on `RequestAgent` and the specified request could not be found or is already destroyed.

Include File

Include the `NativeMessaging_c.hh` file when you use this class.

Chapter 12

Portable Interceptor interfaces and classes

This section describes the BES VisiBroker implementation of *Portable Interceptors* interfaces and classes defined by the OMG Specification. For a complete description of these interfaces and classes, refer to OMG Final Adopted Specification, ptc/2001-04-03, Portable Interceptors.

Note See “Using Portable Interceptors” in the *VisiBroker for C++ Developer's Guide* before using these interfaces.

About Interceptors

The VisiBroker ORB provides a set of APIs known as interceptors which provide a way to plug in additional VisiBroker ORB behavior such as support for transactions and security. Interceptors are hooked into the VisiBroker ORB through which VisiBroker ORB services can intercept the normal flow of execution of the VisiBroker ORB. The following table lists the types of interceptor that VisiBroker supports.

For more information about using portable interceptors, see “Using Portable Interceptors” in the *VisiBroker for C++ Developer's Guide*.

Table 12.1 Types of interceptors VisiBroker supports

Interceptor Type	Description
Portable Interceptor	Portable Interceptors is an OMG standardized feature that allows you to write portable code for interceptors and use it with different vendor ORBs.
Interceptors	Interceptors are Borland Enterprise Server proprietary interceptors defined in VisiBroker.

For more information about using interceptors, see [“5.x Interceptor and object wrapper interfaces and classes” on page 241](#), and “Using Portable Interceptors” in the *VisiBroker for C++ Developer's Guide*.

The following table lists the two types of portable interceptor.

Table 12.2 Types of portable interceptors

Interceptor Type	Description
Request Interceptor	Use to enable VisiBroker ORB services to transfer context information between clients and servers. Request Interceptors are further divided into Client Request Interceptors and Server Request Interceptors.
IOR Interceptors	Use to enable a VisiBroker ORB service to add information, in an IOR, describing the server's or object's ORB service related capabilities. For example, a security service (like SSL) can add its tagged component into the IOR so that clients recognizing that component can establish the connection with the server based on the information in the component.

For more information about using portable interceptors, see “Using Portable Interceptors” in the *VisiBroker for C++ Developer's Guide*.

ClientRequestInfo

```
class PortableInterceptor::ClientRequestInfo : public virtual RequestInfo
```

This class is derived from `RequestInfo`. It is passed to client side interceptors point.

Some methods on `ClientRequestInfo` are not valid at all interception points. The following table shows the validity of each attribute or method. If an attribute is not valid, attempting to access it results in a `BAD_INV_ORDER` being raised with a standard minor code of 14.

Table 12.3 ClientRequestInfo validity

	send_request	send_poll	receive_reply	receive_exception	receive_other
request_id	yes	yes	yes	yes	yes
operation	yes	yes	yes	yes	yes
arguments	yes ¹	no	yes	no	no
exception	yes	no	yes	yes	yes
contexts	yes	no	yes	yes	yes
operation_context	yes	no	yes	yes	yes
result	no	no	yes	no	no
response_expected	yes	yes	yes	yes	yes
sync_scope	yes	no	yes	yes	yes
reply_status	no	no	yes	yes	yes
forward_reference	no	no	no	no	yes ²
get_slot	yes	yes	yes	yes	yes
get_request_service_context	yes	no	yes	yes	yes
get_reply_service_context	no	no	yes	yes	yes
target	yes	yes	yes	yes	yes
effective_target	yes	yes	yes	yes	yes
effective_profile	yes	yes	yes	yes	yes
received_exception	no	no	no	yes	no
received_exception_id	no	no	no	yes	no
get_effective_component	yes	no	yes	yes	yes
get_effective_components	yes	no	yes	yes	yes
get_request_policy	yes	no	yes	yes	yes
add_request_service_context	yes	no	no	no	no

¹ When `ClientRequestInfo` is passed to `send_request()`, there is an entry in the list for every argument, whether in, inout, or out. But only the in and inout arguments will be available.

² If the `reply_status()` does not return `LOCATION_FORWARD`, accessing this attribute raises `BAD_INV_ORDER` with a standard minor code of 14.

Include file

Include the `PortableInterceptor_c.hh` file when you use this class.

ClientRequestInfo methods

```
virtual CORBA::Object_ptr target() = 0;
```

This method returns the object which the client called to perform the operation. See `effective_target()` below.

```
virtual CORBA::Object_ptr effective_target() = 0;
```

This method returns the actual object on which the operation will be invoked. If the `reply_status()` returns `LOCATION_FORWARD`, then on subsequent requests, `effective_target()` will contain the forwarded IOR, while `target` will remain unchanged.

```
virtual IOP::TaggedProfile* effective_profile() = 0;
```

This method returns the profile, in the form of `IOP::TaggedProfile`, that will be used to send the request. If a location forward has occurred for this operation's object and that object's profile changed accordingly, then this profile will be that located profile.

```
virtual CORBA::Any* received_exception() = 0;
```

This method returns the data, in the form of `CORBA::Any`, that contains the exception to be returned to the client.

If the exception is a user exception which cannot be inserted into a `CORBA::Any` (for example, it is unknown or the bindings don't provide the `TypeCode`), then this attribute will be a `CORBA::Any` containing the system exception `UNKNOWN` with a standard minor code of 1. However, the `RepositoryId` of the exception is available in the `received_exception_id` attribute.

```
virtual char* received_exception_id() = 0;
```

This method returns the ID of the `received_exception` to be returned to the client.

```
virtual IOP::TaggedComponent* get_effective_component(CORBA::ULong _id) = 0;
```

This methods returns the `IOP::TaggedComponent` with the given ID from the profile selected for this request.

If there is more than one component for a given component ID, it is undefined which component this operation returns. If there is more than one component for a given component ID, `get_effective_components()` will be called instead.

If no component exists for the given component ID, this operation will raise `BAD_PARAM` with a standard minor code of 28.

Parameter	Description
<code>_id</code>	ID of the component which is to be returned.

```
virtual IOP::TaggedComponentSeq* get_effective_components(CORBA::ULong _id) = 0;
```

This method returns all the tagged components with the given ID from the profile selected for this request. This sequence is in the form of an `IOP::TaggedComponentSeq`.

If no component exists for the given component ID, this operation will raise `BAD_PARAM` with a standard minor code of 28.

Parameter	Description
<code>_id</code>	ID of the components which are to be returned.

```
virtual CORBA::Policy_ptr get_request_policy(CORBA::ULong _type) = 0;
```

This method returns the given policy in effect for this operation.

If the policy type is not valid, either because the specified type is not supported by this ORB or because a policy object of that type is not associated with this Object, `INV_POLICY` with a standard minor code of 2 is raised.

Parameter	Description
<code>_type</code>	Type of policy which specifies the policy to be returned.

```
virtual void add_request_service_context(const IOP::ServiceContext&
_service_context, CORBA::Boolean _replace) = 0;
```

This method allows interceptors to add service contexts to the request.

There is no declaration of the order of the service contexts. They may or may not appear in the order in which they are added.

Parameter	Description
<code>_service_context</code>	<code>IOP::ServiceContext</code> to be added to the request.
<code>_replace</code>	Indicates the behavior of this method when a service context already exists with the given ID. If false, then <code>BAD_INV_ORDER</code> with a standard minor code of 15 is raised. If true, then the existing service context is replaced by the new one.

ClientRequestInterceptor

```
class PortableInterceptor::ClientRequestInterceptor : public virtual Interceptor
```

This `ClientRequestInterceptor` class is used to derive user-defined client side interceptor. A `ClientRequestInterceptor` instance is registered with the VisiBroker ORB (see [“ORBInitializer” on page 227](#) for more information).

Include file

Include the `PortableInterceptor_c.hh` file when you use this class.

ClientRequestInterceptor methods

```
virtual void send_request(ClientRequestInfo_ptr _ri) = 0;
```

This `send_request()` interception point allows an interceptor to query request information and modify the service context before the request is sent to the server.

This interception point may raise a system exception. If it does, no other interceptors' `send_request()` interception points are called. Those interceptors on the Flow Stack are popped and their `receive_exception()` interception points are called.

This interception point may also raise a `ForwardRequest` exception (see [“ForwardRequest” on page 222](#) for more information). If an Interception raises this exception, no other interceptors' `send_request` methods are called. The remaining interceptors in the Flow Stack are popped and have their `receive_other()` interception point called.

Parameter	Description
<code>_ri</code>	<code>ClientRequestInfo</code> instance to be used by interceptor.

```
virtual void send_poll(ClientRequestInfo_ptr _ri) = 0;
```

This `send_poll()` interception point allows an interceptor to query information during a Time-Independent Invocation (TII) polling get reply sequence.

However, as the VisiBroker ORB does not support TII, this `send_poll()` interception point will never be called.

Parameter	Description
<code>_ri</code>	<code>ClientRequestInfo</code> instance to be used by interceptor.

```
virtual void receive_reply(ClientRequestInfo_ptr _ri) = 0;
```

This `receive_reply()` interception point allows an interceptor to query the information on a reply after it is returned from the server and before control is returned to the client.

This interception point may raise a system exception. If it does, no other interceptors' `receive_reply()` methods are called. The remaining interceptors in the Flow Stack will have their `receive_exception()` interception point called.

Parameter	Description
<code>_ri</code>	<code>ClientRequestInfo</code> instance to be used by interceptor.

```
virtual void receive_exception(ClientRequestInfo_ptr _ri) = 0;
```

This `receive_exception()` interception point is called when an exception occurs. It allows an interceptor to query the exception's information before it is raised to the client.

This interception point may raise a system exception. This has the effect of changing the exception which successive interceptors popped from the Flow Stack receive on their calls to `receive_exception()`. The exception raised to the client will be the last exception raised by an interceptor, or the original exception if no interceptor changes the exception.

This interception point may also raise a `ForwardRequest` exception (see [“ForwardRequest” on page 222](#) for more information). If an interceptor raises this exception, no other interceptors' `receive_exception()` interception points are called. The remaining interceptors in the Flow Stack are popped and have their `receive_other()` interception point called.

Parameter	Description
<code>_ri</code>	<code>ClientRequestInfo</code> instance to be used by interceptor.

```
virtual void receive_other(ClientRequestInfo_ptr _ri) = 0;
```

This `receive_other()` interception point allows an interceptor to query the information available when a request results in something other than a normal reply or an exception. For example, a request could result in a retry (for example, a GIOP Reply with a `LOCATION_FORWARD` status was received), or on asynchronous calls, the reply does not immediately follow the request, but control will return to the client and an ending interception point will be called.

For retries, depending on the policies in effect, a new request may or may not follow when a retry has been indicated. If a new request does follow, while this request is a new request, with respect to interceptors, there is one point of correlation between the original request and the retry: because control has not returned to the client, the request scoped `PortableInterceptor::Current` for both the original request and the retrying request is the same (see [“Current” on page 220](#) for more information).

This interception point may raise a system exception. If it does, no other interceptors' `receive_other()` interception points are called. The remaining interceptors in the Flow Stack are popped and have their `receive_exception()` interception point called.

This interception point may also raise a `ForwardRequest` exception (see [“ForwardRequest” on page 222](#) for more information). If an interceptor raises this exception, successive interceptors' `receive_other()` methods are called with the new information provided by the `ForwardRequest` exception.

Parameter	Description
<code>_ri</code>	<code>ClientRequestInfo</code> instance to be used by interceptor.

Codec

```
class IOP::Codec
```

The formats of IOR components and service context data used by ORB services are often defined as CDR encapsulations encoding instances of IDL defined data types. `Codec` provides a mechanism to transfer these components between their IDL data types and their CDR encapsulation representations.

A `Codec` is obtained from the `CodecFactory`. The `CodecFactory` is obtained through a call to `ORB::resolve_initial_references("CodecFactory")`.

Include file

Include the `IOP_c.hh` file when you use this class.

Codec Member Classes

```
class Codec::InvalidTypeForEncoding : public CORBA_UserException
```

This exception is raised by `encode()` or `encode_value()` when an invalid type is specified for the encoding.

```
class Codec::FormatMismatch : public CORBA_UserException
```

This exception is raised by `decode()` or `decode_value()` when the data in the octet sequence cannot be decoded into a `CORBA::Any`.

```
class Codec::TypeMismatch : public CORBA_UserException
```

This exception is raised by `decode_value()` when the given `TypeCode` does not match the given octet sequence.

Codec Methods

```
virtual CORBA::OctetSequence* encode(const CORBA::Any& _data) = 0;
```

This method converts the given data in the form of a `CORBA::Any` into an octet sequence based on the encoding format effective for this `Codec`. This octet sequence contains both the `TypeCode` and the data of the type.

This operation may raise `InvalidTypeForEncoding`.

Parameter	Description
<code>_data</code>	Data, in the form of a <code>CORBA::Any</code> , to be encoded into an octet sequence.

```
virtual CORBA::Any* decode(const CORBA::OctetSequence& _data) = 0;
```

This method decodes the given octet sequence into a `CORBA::Any` object based on the encoding format effective for this `Codec`.

This method raises `FormatMismatch` if the octet sequence cannot be decoded into a `CORBA::Any`.

Parameter	Description
<code>_data</code>	Data, in the form of an octet sequence, to be decoded into a <code>CORBA::Any</code> .

```
virtual CORBA::OctetSequence* encode_value(const CORBA::Any& _data) = 0;
```

This method converts the given `CORBA::Any` object into an octet sequence based on the encoding format effective for this `Codec`. Only the data from the `CORBA::Any` is encoded, not the `TypeCode`.

This operation may raise `InvalidTypeForEncoding`.

Parameter	Description
<code>_data</code>	Octet sequence containing the data from the encoded <code>CORBA::Any</code> .

```
virtual CORBA::Any* decode_value(const CORBA::OctetSequence& _data,
CORBA::TypeCode_ptr _tc) = 0;
```

This method decodes the given octet sequence into a `CORBA::Any` based on the given `TypeCode` and the encoding format effective for this `Codec`.

This method raises `FormatMismatch` if the octet sequence cannot be decoded into a `CORBA::Any`.

Parameter	Description
<code>_data</code>	Data, in the form of an octet sequence, to be decoded into a <code>CORBA::Any</code> .
<code>_tc</code>	<code>TypeCode</code> to be used to decode the data.

CodecFactory

```
class IOP::CodecFactory
```

This class is used to obtain a `Codec`. The `CodecFactory` is obtained through a call to `ORB::resolve_initial_references("CodecFactory")`.

Include file

Include the `IOP_c.hh` file when you use this class.

CodecFactory Member

```
class CodecFactory::UnknownEncoding : public CORBA_UserException
```

This exception is raised if `CodecFactory` cannot create a `Codec`. See `create_codec()` function below.

CodecFactory Method

```
virtual Codec_ptr create_codec(const Encoding& _enc) = 0;
```

This `create_codec()` method creates a `Codec` of the given encoding.

This method raises `UnknownEncoding` if this factory cannot create a `Codec` of the given encoding.

Parameter	Description
<code>_enc</code>	Specifies the encoding to be used for creating a <code>Codec</code> .

Current

```
class PortableInterceptor::Current: public virtual CORBA::Current, public virtual
CORBA_Object
```

The `Current` class is merely a slot table, the slots of which are used by each service to transfer their context data between their context and the request's or reply's service context.

Each service which wishes to use `Current` reserves a slot or slots at initialization time (see “[virtual CORBA::ULong allocate_slot_id\(\) = 0;](#)” on page 230 for more information) and uses those slots during the processing of requests and replies.

Before an invocation is made, `Current` is obtained by way of a call to `ORB::resolve_initial_references("PICurrent")`.

From within the interception points, the data on `Current` that has moved from the thread scope to the request scope is available by way of the `get_slot()` method on the `RequestInfo` object. A `Current` can still be obtained by way of `resolve_initial_references()`, but that is the interceptor's thread scope `Current`.

Include file

Include the `PortableInterceptor_c.hh` file when you use this class.

Current Methods

```
virtual CORBA::Any* get_slot(CORBA::ULong _id);
```

A service can get the slot data it sets in `PICurrent` by way of the `get_slot()` method. The data is in the form of a `CORBA::Any` object.

If the given slot has not been set, a `CORBA::Any` containing a type code with a `TCKind` value of `tk_null`, no value is returned.

If `get_slot()` is called on a slot that has not been allocated, `InvalidSlot` is raised.

If `get_slot()` is called from within an ORB initializer (see “[ORBInitializer](#)” on [page 227](#) for more information), `BAD_INV_ORDER` with a minor code of 14 is raised.

Parameter	Description
<code>_id</code>	SlotId of the slot from which the data will be returned.

```
virtual void set_slot(CORBA::ULong _id, const CORBA::Any& _data);
```

A service sets data in a slot with `set_slot()`. The data is in the form of a `CORBA::Any` object.

If data already exists in that slot, it is overridden.

If `set_slot()` is called on a slot that has not been allocated, `InvalidSlot` is raised.

If `set_slot()` is called from within an ORB initializer (see “[ORBInitializer](#)” on [page 227](#) for more information) `BAD_INV_ORDER` with a minor code of 14 is raised.

Parameter	Description
<code>_id</code>	SlotId of the slot from which the data will be set.
<code>_data</code>	data, in the form of a <code>CORBA::Any</code> object, which will be set to the identified slot.

Encoding

```
struct IOP::Encoding
```

This structure defines the encoding format of a `Codec`. It details the encoding format, such as CDR Encapsulation encoding, and the major and minor versions of that format.

The supported encodings are:

- `ENCODING_CDR_ENCAPS`, version 1.0;
- `ENCODING_CDR_ENCAPS`, version 1.1;
- `ENCODING_CDR_ENCAPS`, version 1.2;
- `ENCODING_CDR_ENCAPS` for all future versions of GIOP as they arise.

Include file

Include the `IOP_c.hh` file when you use this struct.

Members

`CORBA::Short format;`

This member holds the encoding format for a Codec.

`CORBA::Octet major_version;`

This member holds the major version number for a Codec.

`CORBA::Octet minor_version;`

This member holds minor version number for a Codec.

ExceptionList

`class Dynamic::ExceptionList`

Use this class to hold exceptions information returned from the method `exceptions()` in the class `RequestInfo`. It is an implementation of variable-length array of type `CORBA::TypeCode`. The length of `ExceptionList` is available at run time.

For more information, see “[virtual Dynamic::ExceptionList* exceptions\(\) = 0;](#)” on [page 233](#).

Include file

Include the `Dynamic_c.hh` file when you use this class.

ForwardRequest

`class PortableInterceptor::ForwardRequest : public CORBA_UserException`

The `ForwardRequest` exception is the means by which an interceptor can indicate to the ORB that a retry of the request should occur with the new object given in the exception. This behavior of causing a retry only occurs if the ORB receives a `ForwardRequest` from an interceptor. If `ForwardRequest` is raised anywhere else, it is passed through the ORB as is normal for a user exception.

If an interceptor raises a `ForwardRequest` exception in response to a call of an interceptor, no other interceptors are called for that interception point. The remaining interceptors in the Flow Stack will have their appropriate ending interception point called: `receive_other()` on the client, or `send_other()` on the server. The `reply_status()` in the `receive_other()` or `send_other()` will return `LOCATION_FORWARD`.

Include file

Include the `PortableInterceptor_c.hh` file when you use this class.

Interceptor

```
class PortableInterceptor::Interceptor
```

This is the base class from which all interceptors are derived.

Include file

Include the `PortableInterceptor_c.hh` file when you use this class.

Interceptor methods

```
virtual char* name() = 0;
```

This method returns the name of the interceptor. Each interceptor may have a name which can be used to order the lists of interceptors. Only one interceptor of a given name can be registered with the VisiBroker ORB for each interceptor type. An interceptor may be anonymous, such as it has an empty string as the `name` attribute. Any number of anonymous interceptors may be registered with the VisiBroker ORB.

```
virtual void destroy() = 0;
```

This method is called during `ORB::destroy()`. When `ORB::destroy()` is called by an application, the VisiBroker ORB:

- 1 waits for all requests in progress to complete
- 2 calls the `Interceptor::destroy()` method for each interceptor
- 3 completes destruction of the ORB

Method invocations from within `Interceptor::destroy()` on object references for objects implemented on the ORB being destroyed result in undefined behavior. However, method invocations on objects implemented on VisiBroker ORB, other than the one being destroyed, are permitted. (This means that the VisiBroker ORB being destroyed is still capable of acting as a client, but not as a server.)

IORInfo

```
class PortableInterceptor::IORInfo
```

The `IORInfo` interface provides the server side ORB service with access to the applicable policies during IOR construction and the ability to add components. The ORB passes an instance of its implementation of this interface as a parameter to `IORInterceptor::establish_components()`.

The table below defines the validity of each attribute or method in `IORInfo` in the methods defined in the `IORInterceptor`.

Table 12.4 IORInfo validity

	<code>establish_components</code>	<code>components_established</code>
<code>get_effective_policy</code>	yes	yes
<code>add_component</code>	yes	no
<code>add_component_to_profile</code>	yes	no
<code>manager_id</code>	yes	yes
<code>state</code>	yes	yes

Table 12.4 IORInfo validity

	establish_components	components_established
adapter_template	no	yes
current_factory	no	yes

If an illegal call is made to an attribute or method in IORInfo, the `BAD_INV_ORDER` system exception is raised with a standard minor code value of 14.

Include file

Include the `PortableInterceptor_c.hh` file when you use this class.

IORInfo Methods

```
virtual CORBA::Policy_ptr get_effective_policy(CORBA::ULong _type) = 0;
```

An ORB service implementation may determine what server side policy of a particular type is in effect for an IOR being constructed by calling the `get_effective_policy()` method. When the IOR being constructed is for an object implemented using a POA, all Policy objects passed to the `PortableServer::POA::create_POA()` call that created that POA are accessible via `get_effective_policy`.

If a policy for the given type is not known to the ORB, then this method will raise `INV_POLICY` with a standard minor code of 3.

Parameter	Description
<code>_type</code>	<code>CORBA::PolicyType</code> specifying the type of policy to return.

```
virtual void add_ior_component(const IOP::TaggedComponent& _a_component) = 0;
```

This method is called from `establish_components()` to add a tagged component to the set which will be included when constructing IORs. The components in this set will be included in all profiles.

Any number of components may exist with the same component ID.

Parameter	Description
<code>_a_component</code>	<code>IOP::TaggedComponent</code> to be added.

```
virtual void add_ior_component_to_profile(const IOP::TaggedComponent&
_a_component, CORBA::ULong _profile_id) = 0;
```

This method is called from `establish_components()` to add a tagged component to the set which will be included when constructing IORs. The components in this set will be included in the specified profile.

Any number of components may exist with the same component ID.

If the given profile ID does not define a known profile or it is impossible to add components to that profile, `BAD_PARAM` is raised with a standard minor code of 29.

Parameter	Description
<code>_a_component</code>	<code>IOP::TaggedComponent</code> to be added.
<code>_profile_id</code>	<code>IOP::ProfileId</code> of the profile to which this component will be added.


```
virtual CORBA::Long manager_id() = 0;
```

This method returns the attribute that provides an opaque handle to the manager of the adapter. This is used for reporting state changes in adapters managed by the same adapter manager.

```
virtual CORBA::Short state() = 0;
```

This method returns the current state of the adapter. This must be one of HOLDING, ACTIVE, DISCARDING, INACTIVE, NON_EXISTENT.

```
virtual ObjectReferenceTemplate_ptr adapter_template() = 0;
```

This method returns the attribute that provides a means to obtain an object reference template whenever an IOR interceptor is invoked. There is no standard way to directly create an object reference template. The value of `adapter_template()` returns is the template created for the adapter policies and IOR interceptor calls to `add_component()` and `add_component_to_profile()`. The value of the `adapter_template()` returns is never changed for the lifetime of the object adapter.

```
virtual ObjectReferenceFactory_ptr current_factory() = 0;
```

This method returns the attribute that provides access to the factory that will be used by the adapter to create object references. `current_factory()` initially has the same value as the `adapter_template` attribute, but this can be changed by setting `current_factory` to another factory. All object references created by the object adapter must be created by calling the `make_object()` method on `current_factory`.

```
virtual void current_factory(ObjectReferenceFactory_ptr _current_factory) = 0;
```

This method sets the `current_factory` attribute. The value of the `current_factory` attribute that is used by the adapter can only be set during the call to the `components_established` method.

Parameter	Description
<code>_current_factory</code>	<code>current_factory</code> object which is to be set.

IORInfoExt

```
class IORInfoExt: public PortableInterceptor::IORInfo
```

This is the VisiBroker extensions to Portable Interceptors to allow installing of a POA scoped Server Request Interceptor. This IORInfoExt interface is inherited from IORInfo interface and has additional methods to support POA scoped Server Request Interceptor.

Include file

Include the `PortableInterceptorExt_c.hh` file when you use this class.

IORInfoExt Methods

```
virtual void add_server_request_interceptor (ServerRequestInterceptor_ptr
_interceptor) = 0;
```

This method is used to add a POA-scoped (not an ORB-scoped) server side request interceptor to a service.

Parameter	Description
<code>_interceptor</code>	ServerRequestInterceptor to be added.

```
virtual char* full_poa_name();
```

This method return the full POA name.

IORInterceptor

```
class PortableInterceptor::IORInterceptor : public virtual Interceptor
```

In some cases, a portable ORB service implementation may need to add information describing the server's or object's ORB service related capabilities to object references in order to enable the ORB service implementation in the client to function properly.

This is supported through the `IORInterceptor` and `IORInfo` interfaces.

The IOR Interceptor is used to establish tagged components in the profiles within an IOR.

Include file

Include the `PortableInterceptor_c.hh` file when you use this class.

IORInterceptor Methods

```
virtual void establish_components(IORInfo_ptr _info) = 0;
```

A server side ORB calls the `establish_components()` method on all registered `IORInterceptor` instances when it is assembling the list of components that will be included in the profile(s) of an object reference. This method is not necessarily called for each individual object reference. In the case of the POA, these calls are made each time `POA::create_POA()` is called. In other adapters, these calls would typically be made when the adapter is initialized. The adapter template is not available at this stage since information (the components) needed in the adapter template is being constructed.

Parameter	Description
<code>_info</code>	<code>IORInfo</code> instance used by the ORB service to query applicable policies and add components to be included in the generated IORs.

```
virtual void components_established(IORInfo_ptr _info) = 0;
```

After all of the `establish_components()` methods have been called, the `components_established()` methods are invoked on all registered IOR interceptors. The adapter template is available at this stage. The `current_factory` attribute may be get or set at this stage.

Any exception that occurs in `components_established()` is returned to the caller of `components_established()`. In the case of the POA, this causes the `create_POA` call to fail, and an `OBJ_ADAPTER` exception with a standard minor code of 6 is returned to the invoker of `create_POA()`.

Parameter	Description
<code>_info</code>	IORInfo instance used by the ORB service to access applicable policies.

```
virtual void adapter_manager_state_changed(CORBA::Long _id, CORBA::Short _state) = 0;
```

Any time the state of an adapter manager changes, the `adapter_manager_state_changed()` method is invoked on all registered IOR interceptors.

If a state change is reported through `adapter_manager_state_changed()`, it is not reported through `adapter_state_changed()`.

Parameter	Description
<code>_id</code>	IORInfo instance used by the ORB service to access applicable policies.
<code>_state</code>	new state of the object adapter.

```
virtual void adapter_state_changed(const ObjectReferenceTemplateSeq& _templates, CORBA::Short _state) = 0;
```

Object adapter state changes are reported to this method any time the state of one or more adapters changes for reasons unrelated to adapter manager state changes. The `templates` argument identifies the object adapters that have changed state by the template ID information. The sequence contains the adapter templates for all object adapters that have made the state transition being reported.

Parameter	Description
<code>_templates</code>	identifies the object adapters that have changed state by the template ID information.
<code>_state</code>	new state of the object adapter.

ORBInitializer

```
class PortableInterceptor::ORBInitializer
```

An interceptor is registered by registering an associated `ORBInitializer` object which implements the `ORBInitializer` class. When an ORB is initializing, it calls each registered `ORBInitializer`, passing it an `ORBInitInfo` object which is used to register its interceptor.

Include file

Include the `PortableInterceptor_c.hh` file when you use this class.

ORBInitializer Methods

```
virtual void pre_init(ORBInitInfo_ptr _info) = 0;
```

This method is called during ORB initialization. If it is expected that initial services registered by an interceptor will be used by other interceptors, then those initial services are registered at this point via calls to

```
ORBInitInfo::register_initial_reference().
```

Parameter	Description
<code>_info</code>	an object that provides initialization attributes and methods by which interceptors can be registered.

```
virtual void post_init(ORBInitInfo_ptr _info) = 0;
```

This method is called during ORB initialization. If a service must resolve initial references as part of its initialization, it can assume that all initial references will be available at this point.

Calling the `post_init()` methods is not the final task of ORB initialization. The final task, following the `post_init()` calls, is attaching the lists of registered interceptors to the ORB. Therefore, the ORB does not contain the interceptors during calls to `post_init()`. If an ORB-mediated call is made from within `post_init()`, no request interceptors will be invoked on that call. Likewise, if a method is performed which causes an IOR to be created, no IOR interceptors will be invoked.

Parameter	Description
<code>_info</code>	An object that provides initialization attributes and methods by which interceptors can be registered.

ORBInitInfo

```
class PortableInterceptor::ORBInitInfo
```

This `ORBInitInfo` class is passed to `ORBInitializer` object for registering interceptors.

Include file

Include the `PortableInterceptor_c.hh` file when you use this class.

ORBInitInfo Member Classes

```
class DuplicateName : public CORBA_UserException;
```

Only one interceptor of a given name can be registered with the ORB for each interceptor type. If an attempt is made to register a second interceptor with the same name, `DuplicateName` is raised.

An interceptor may be anonymous, such as it has an empty string as the name attribute. Any number of anonymous interceptors may be registered with the ORB, so if the interceptor being registered is anonymous, the registration operation will not raise `DuplicateName`.

```
class InvalidName: public CORBA_UserException
```

This exception is raised by `register_initial_reference()` and `resolve_initial_references()`.

`register_initial_reference()` raises `InvalidName` if:

This method is called with an empty string id; or

This method is called with an id that is already registered, including the default names defined by OMG.

`resolve_initial_references()` raises `InvalidName` if the name to be resolved is invalid.

ORBInitInfo Methods

```
virtual CORBA::StringSequence* arguments() = 0;
```

This method returns the arguments passed to `ORB_init()`. They may or may not contain the ORB's arguments.

```
virtual char* orb_id() = 0;
```

This method returns the ID of the ORB being initialized.

```
virtual IOP::CodecFactory_ptr codec_factory() = 0;
```

This method returns the `IOP::CodecFactory`. The `CodecFactory` is normally obtained via a call to `ORB::resolve_initial_references("CodecFactory")`, but since the ORB is not yet available and interceptors, particularly when processing service contexts, will require a `Codec`, a means of obtaining a `Codec` is necessary during ORB initialization.

```
virtual void register_initial_reference(const char* _id, CORBA::Object_ptr _obj) = 0;
```

If this method is called with an id, "Y", and an object, YY, then a subsequent call to `register_initial_reference()` will return object YY.

This method is identical to `ORB::register_initial_reference()`. This same functionality exists here because the ORB, not yet fully initialized, is not yet available but initial references may need to be registered as part of Interceptor registration. The only difference is that the version of this method on the ORB uses PIDL (`CORBA::ORB::ObjectId` and `CORBA::ORB::InvalidName`) whereas the version in this interface uses IDL defined in this interface; the semantics are identical.

`register_initial_reference()` raises `InvalidName` if

This method is called with an empty string id; or

This method is called with an id that is already registered, including the default names defined by OMG.

Parameter	Description
<code>_id</code>	ID by which the initial reference will be known.
<code>_obj</code>	The initial reference itself.

```
virtual CORBA::Object_ptr resolve_initial_references(const char* _id) = 0;
```

This method is only valid during `post_init()`. It is identical to `ORB::resolve_initial_references()`. This same functionality exists here because the ORB, not yet fully initialized, is not yet available but initial references may be required from the ORB as part of Interceptor registration.

Parameter	Description
<code>_id</code>	ID by which the initial reference will be known.

If the name to be resolved is invalid, `resolve_initial_references()` will raise `InvalidName`.

```
virtual void add_client_request_interceptor(ClientRequestInterceptor_ptr
_interceptor) = 0;
```

This method is used to add a client side request interceptor to the list of client side request interceptors.

If a client side request interceptor has already been registered with this interceptor's name, `DuplicateName` will be raised.

Parameter	Description
<code>_interceptor</code>	<code>ClientRequestInterceptor</code> to be added.

```
virtual void add_server_request_interceptor(ServerRequestInterceptor_ptr
_interceptor) = 0;
```

This method is used to add a server side request interceptor to the list of server side request interceptors.

If a server side request interceptor has already been registered with this interceptor's name, `DuplicateName` is raised.

Parameter	Description
<code>_interceptor</code>	<code>ServerRequestInterceptor</code> to be added.

```
virtual void add_ior_interceptor(IORInterceptor_ptr _interceptor) = 0;
```

This method is used to add an IOR interceptor to the list of IOR interceptors.

If an IOR interceptor has already been registered with this interceptor's name, `DuplicateName` is raised.

Parameter	Description
<code>_interceptor</code>	<code>IORInterceptor</code> to be added.

```
virtual CORBA::ULong allocate_slot_id() = 0;
```

Returns the index to the slot which has been allocated.

A service calls `allocate_slot_id` to allocate a slot on `PortableInterceptor::Current`.

Note While slot id's can be allocated within an ORB initializer, the slots themselves cannot be initialized. Calling `set_slot()` or `get_slot()` on the `Current` (see [“Current” on page 220](#) for more information) within an ORB initializer will raise a `BAD_INV_ORDER` with a minor code of 14.

```
virtual void register_policy_factory(CORBA::ULong _type, PolicyFactory_ptr
_policy_factory) = 0;
```

This method registers a `PolicyFactory` for the given `PolicyType`.

If a `PolicyFactory` already exists for the given `PolicyType`, `BAD_INV_ORDER` is raised with a standard minor code of 16.

Parameter	Description
<code>_type</code>	CORBA::PolicyType that the given <code>PolicyFactory</code> serves.
<code>_policy_factory</code>	factory for the given <code>CORBA::PolicyType</code> .

Parameter

```
struct Dynamic::Parameter
```

This structure holds the parameter information. This structure is the element used in `ParameterList` (see “[ParameterList](#)” on page 231 for more information).

Include file

Include the `Dynamic_c.hh` file when you use this struct.

Members

```
CORBA::Any argument;
```

This member stores the parameter data in the form of `CORBA::Any`.

```
CORBA::ParameterMode mode;
```

This member specifies the mode of a parameter. Its value can be one of the enum values: `PARAM_IN`, `PARAM_OUT` or `PARAM_INOUT`.

ParameterList

```
class Dynamic::ParameterList
```

This class is used to pass parameters information returned from the method `arguments()` in the class `RequestInfo`. It is an implementation of variable-length array of type `Parameter`. The length of `ParameterList` is available at run-time.

For more information, see “[virtual Dynamic::ParameterList* arguments\(\) = 0;](#)” on page 233.

Include file

Include the `Dynamic_c.hh` file when you use this class.

PolicyFactory

```
class PortableInterface::PolicyFactory
```

A portable ORB service implementation registers an instance of the `PolicyFactory` interface during ORB initialization. The POA is required to preserve any policy which is registered with `ORBInitInfo` in this manner.

Include file

Include the `PortableInterceptor_c.hh` file when you use this class.

PolicyFactory Method

```
virtual CORBA::Policy_ptr create_policy(CORBA::ULong _type, const CORBA::Any&
_value) = 0;
```

The ORB calls `create_policy()` on a registered `PolicyFactory` instance when `CORBA::ORB::create_policy()` is called for the `PolicyType` under which the `PolicyFactory` has been registered. The `create_policy()` method then returns an instance of the appropriate interface derived from `CORBA::Policy` whose value corresponds to the specified `CORBA::Any`. If it cannot, it will raise an exception as described for `CORBA::ORB::create_policy()`.

Parameter	Description
<code>_type</code>	A <code>CORBA::PolicyType</code> specifying the type of policy being created.
<code>_value</code>	An <code>CORBA::Any</code> containing data with which to construct the <code>CORBA::Policy</code> .

RequestInfo

```
class PortableInterceptor::RequestInfo
```

This is the base class from which `ClientRequestInfo` and `ServerRequestInfo` are derived. Each interception point is given an object through which the `Interceptor` can access request information. client side and server side interception points are concerned with different information, so there are two information objects: `ClientRequestInfo` is passed to the client side interception points and `ServerRequestInfo` is passed to the server side interception points. But there is information that is common to both, so they both inherit from this common interface: `RequestInfo`.

Include file

Include the `PortableInterceptor_c.hh` file when you use this class.

RequestInfo methods

```
virtual CORBA::ULong request_id() = 0;
```

This method returns the ID which uniquely identifies an active request / reply sequence. Once a request / reply sequence is concluded this ID may be reused.

Note This ID is not the same as the GIOP `request_id`. If GIOP is the transport mechanism used, then these IDs may very well be the same, but this is not guaranteed nor required.

```
virtual char* operation() = 0;
```

This method returns name of the operation being invoked.

```
virtual Dynamic::ParameterList* arguments() = 0;
```

This method returns a `Dynamic::ParameterList` containing the arguments on the operation being invoked. If there are no arguments, this attribute will be a zero length sequence.

```
virtual Dynamic::ExceptionList* exceptions() = 0;
```

This method returns a `Dynamic::ExceptionList` describing the `TypeCodes` of the user exceptions that this operation invocation may raise. If there are no user exceptions, this attribute will be a zero length sequence.

```
virtual CORBA::StringSequence* contexts() = 0;
```

This method returns a `CORBA::StringSequence` describing the contexts that may be passed on this operation invocation. If there are no contexts, this attribute will be a zero length sequence.

```
virtual CORBA::StringSequence* operation_context() = 0;
```

This method returns a `CORBA::StringSequence` containing the contexts being sent on the request.

```
virtual CORBA::Any* result() = 0;
```

This method returns the data, in the form of `CORBA::Any`, that contains the result of the operation invocation. If the operation return type is void, this attribute will be a `CORBA::Any` containing a type code with a `TCKind` value of `tk_void` and no value.

```
virtual CORBA::Boolean response_expected() = 0;
```

This method returns a boolean value which indicates whether a response is expected.

On the client, a reply is not returned when `response_expected()` is false, so `receive_reply()` cannot be called. `receive_other()` is called unless an exception occurs, in which case `receive_exception()` is called.

```
virtual CORBA::Short sync_scope() = 0;
```

This method returns an attribute, defined in the Messaging specification, is pertinent **only** when `response_expected()` is false. If `response_expected()` is true, the value of `sync_scope()` is undefined. It defines how far the request will progress before control is returned to the client. This attribute may have one of the following values:

- `Messaging::SYNC_NONE`
- `Messaging::SYNC_WITH_TRANSPORT`
- `Messaging::SYNC_WITH_SERVER`
- `Messaging::SYNC_WITH_TARGET`

On the server, for all scopes a reply will be created from the return of the target operation call, but the reply will not return to the client. Although it does not return to the client, it does occur, so the normal server side interception points are followed (for example, `receive_request_service_contexts()`, `receive_request()`, `send_reply()` or `send_exception()`).

For `SYNC_WITH_SERVER` and `SYNC_WITH_TARGET`, the server does send an empty reply back to the client before the target is invoked. This reply is not intercepted by server side Interceptors.

```
virtual CORBA::Short reply_status() = 0;
```

This method returns an attribute which describes the state of the result of the operation invocation. Its value can be one of the following:

- `PortableInterceptor::SUCCESSFUL = 0`
- `PortableInterceptor::SYSTEM_EXCEPTION = 1`
- `PortableInterceptor::USER_EXCEPTION = 2`
- `PortableInterceptor::LOCATION_FORWARD = 3`
- `PortableInterceptor::TRANSPORT_RETRY = 4`

On the client:

- Within the `receive_reply` interception point, this attribute will only be `SUCCESSFUL`.
- Within the `receive_exception` interception point, this attribute will be either `SYSTEM_EXCEPTION` or `USER_EXCEPTION`.
- Within the `receive_other` interception point, this attribute will be any of: `SUCCESSFUL`, `LOCATION_FORWARD`, or `TRANSPORT_RETRY`. `SUCCESSFUL` means an asynchronous request returned successfully. `LOCATION_FORWARD` means that a reply came back with `LOCATION_FORWARD` as its status. `TRANSPORT_RETRY` means that the transport mechanism indicated a retry—a GIOP reply with a status of `NEEDS_ADDRESSING_MODE`, for instance.

On the server:

- Within the `send_reply` interception point, this attribute will only be `SUCCESSFUL`.
- Within the `send_exception` interception point, this attribute will be either `SYSTEM_EXCEPTION` or `USER_EXCEPTION`.
- Within the `send_other` interception point, this attribute will be any of: `SUCCESSFUL`, or `LOCATION_FORWARD`. `SUCCESSFUL` means an asynchronous request returned successfully. `LOCATION_FORWARD` means that a reply came back with `LOCATION_FORWARD` as its status.

```
virtual CORBA::Object_ptr forward_reference() = 0;
```

If the `reply_status()` returns `LOCATION_FORWARD`, then this method returns an object to which the request will be forwarded. It is indeterminate whether a forwarded request will actually occur.

```
virtual CORBA::Any* get_slot(CORBA::ULong _id) = 0;
```

This method returns the data, in the form of a `CORBA::Any`, from the given slot of the `PortableInterceptor::Current` that is in the scope of the request.

If the given slot has not been set, then a `CORBA::Any` containing a type code with a `TCKind` value of `tk_null` is returned.

If the ID does not define an allocated slot, `InvalidSlot` is raised.

See “[Current](#)” on page 220 for an explanation of slots and the `PortableInterceptor::Current`.

Parameter	Description
<code>_id</code>	SlotId of the slot which is to be returned.

```
virtual IOP::ServiceContext* get_request_service_context(CORBA::ULong _id) = 0;
```

This method returns a copy of the service context with the given ID that is associated with the request.

If the request's service context does not contain an entry for that ID, `BAD_PARAM` with a standard minor code of 26 is raised.

Parameter	Description
<code>_id</code>	<code>IOP::ServiceContext</code> of the slot which is to be returned.

```
virtual IOP::ServiceContext* get_reply_service_context(CORBA::ULong _id) = 0;
```

This method returns a copy of the service context with the given ID that is associated with the reply.

If the request's service context does not contain an entry for that ID, `BAD_PARAM` with a standard minor code of 26 is raised.

Parameter	Description
<code>_id</code>	<code>IOP::ServiceContext</code> of the slot which is to be returned.

ServerRequestInfo

```
class PortableInterceptor::ServerRequestInfo : public virtual RequestInfo
```

This class is derived from `RequestInfo`. It is passed to server side interception points.

Some methods on `ServerRequestInfo` are not valid at all interception points. The table below shows the validity of each attribute or method. If it is not valid, attempting to access it will result in a `BAD_INV_ORDER` being raised with a standard minor code of 14.

Table 12.5 ServerRequestInfo

	receive_request_service_contexts	receive_request	send_reply	send_exception	send_other
request_id	yes	yes	yes	yes	yes
operation	yes	yes	yes	yes	yes
arguments	no	yes ¹	yes	no ²	no ²
exception	no	yes	yes	yes	yes
contexts	no	yes	yes	yes	yes
operation_context	no	yes	yes	no	no

Table 12.5 ServerRequestInfo

	receive_request_service_contexts	receive_request	send_reply	send_exception	send_other
result	no	no	yes	no	no
response_expected	yes	yes	yes	yes	yes
sync_scope	yes	yes	yes	yes	yes
reply_status	no	no	yes	yes	yes
forward_reference	no	no	no	no	yes ²
get_slot	yes	yes	yes	yes	yes
get_request_service_context	yes	yes	yes	yes	yes
get_reply_service_context	no	no	yes	yes	yes
sending_exception	no	no	no	yes	no
object_id	no	yes	yes	yes ³	yes ³
adapter_id	no	yes	yes	yes ³	yes ³
server_id	no	yes	yes	yes	yes
orb_id	no	yes	yes	yes	yes
adapter_name	no	yes	yes	yes	yes
target_most_derived_interface	no	yes	no ⁴	no ⁴	no ⁴
get_server_policy	yes	yes	yes	yes	yes
set_slot	yes	yes	yes	yes	yes
target_is_a	no	yes	no ⁴	no ⁴	no ⁴
add_reply_service_context	yes	yes	yes	yes	

¹ When `ServerRequestInfo` is passed to `receive_request()`, there is an entry in the list for every argument, whether in, inout, or out. But only the in and inout arguments will be available.

² If the `reply_status()` does not return `LOCATION_FORWARD`, accessing this attribute will raise `BAD_INV_ORDER` with a standard minor code of 14.

³ If the servant locator caused a location forward, or raised an exception, this attribute / method may not be available in this interception point. `NO_RESOURCES` with a standard minor code of 1 will be raised if it is not available.

⁴ The method is not available in this interception point because the necessary information requires access to the target object's servant, which may no longer be available to the ORB. For example, if the object's adapter is a POA that uses a `ServantLocator`, then the ORB invokes the interception point after it calls `ServantLocator::postinvoke()`.

Include file

Include the `PortableInterceptor_c.hh` file when you use this class.

ServerRequestInfo methods

```
virtual CORBA::Any* sending_exception() = 0;
```

This method returns data, in the form `CORBA::Any`, that contains the exception to be returned to the client.

If the exception is a user exception which cannot be inserted into a `CORBA::Any` (e.g., it is unknown or the bindings don't provide the `TypeCode`), then this attribute will be an `CORBA::Any` containing the system exception `UNKNOWN` with a standard minor code of 1.

```
virtual char* server_id() = 0;
```

This method returns the value that was passed into the `ORB::init` call using the `-ORBServerId` argument when the ORB was created.

```
virtual char* orb_id() = 0;
```

The method returns the value that was passed into the `ORB::init()` call.

In Java, this is accomplished using the `-ORBid` argument in the `ORB.init` call that created the ORB containing the object adapter that created this template. What happens if the same `ORBid` is used on multiple `ORB::init()` calls in the same server is currently undefined.

```
virtual CORBA::StringSequence* adapter_name() = 0;
```

The method returns the name for the object adapter, in the form of `CORBA::StringSequence`, that services requests for the invoked object. In the case of the POA, the `adapter_name` is the sequence of names from the root POA to the POA that services the request. The root POA is not named in this sequence.

```
virtual CORBA::OctetSequence* object_id() = 0;
```

This method returns the opaque `object_id`, in the form of `CORBA::OctetSequence`, that describes the target of the operation invocation.

```
virtual CORBA::OctetSequence* adapter_id() = 0;
```

This method returns opaque identifier for the object adapter, in the form of `CORBA::OctetSequence`.

```
virtual char* target_most_derived_interface() = 0;
```

This method returns the `RepositoryID` for the most derived interface of the servant.

```
virtual CORBA::Policy_ptr get_server_policy(CORBA::ULong _type) = 0;
```

This method returns the policy in effect for this operation for the given policy type. The returned `CORBA::Policy` object will only be a policy whose type was registered via `register_policy_factory()`.

If a policy for the given type was not registered via `register_policy_factory`, this method will raise `INV_POLICY` with a standard minor code of 3.

Parameter	Description
<code>_type</code>	The <code>CORBA::PolicyType</code> which specifies the policy to be returned.

```
virtual void set_slot(CORBA::ULong _id, const CORBA::Any& _data) = 0;
```

This method allows an `Interceptor` to set a slot in the `PortableInterceptor::Current` that is in the scope of the request. If data already exists in that slot, it will be overwritten.

If the ID does not define an allocated slot, `InvalidSlot` is raised.

See [“Current” on page 220](#) for an explanation of slots and `PortableInterceptor::Current`.

Parameter	Description
<code>_id</code>	The <code>SlotId</code> of the slot.
<code>_data</code>	The data, in the form of a <code>CORBA::Any</code> , to store in that slot.

```
virtual CORBA::Boolean target_is_a(const char* _id) = 0;
```

This method returns true if the servant is the given `RepositoryId`, false if it is not.

Parameter	Description
<code>_id</code>	The caller wants to know if the servant is this <code>CORBA::RepositoryId</code> .

```
virtual void add_reply_service_context(const IOP::ServiceContext&  
_service_context, CORBA::Boolean _replace) = 0;
```

This method allows Interceptors to add service contexts to the request.

There is no declaration of the order of the service contexts. They may or may not appear in the order that they are added.

Parameter	Description
<code>_service_context</code>	The <code>IOP::ServiceContext</code> to add to the reply.
<code>_replace</code>	Indicates the behavior of this method when a service context already exists with the given ID. If <code>false</code> , then <code>BAD_INV_ORDER</code> with a standard minor code of 15 is raised. If <code>true</code> , then the existing service context is replaced by the new one.

ServerRequestInterceptor

```
class PortableInterceptor::ServerRequestInterceptor : public virtual Interceptor
```

This `ServerRequestInterceptor` class is used to derive user-defined server side interceptor. A `ServerRequestInterceptor` instance is registered with the ORB (see [“ORBInitializer” on page 227](#) for more information).

Include file

Include the `PortableInterceptor_c.hh` file when you use this class.

ServerRequestInterceptor methods

```
virtual void receive_request_service_contexts(ServerRequestInfo_ptr _ri) = 0;
```

At this `receive_request_service_contexts()` interception point, Interceptors must get their service context information from the incoming request and transfer it to `PortableInterceptor::Current`'s slots.

This interception point is called before the servant manager is called. Operation parameters are not yet available at this point. This interception point may or may not execute in the same thread as the target invocation.

This interception point may raise a system exception. If it does, no other Interceptors' `receive_request_service_contexts()` interception points are called. Those Interceptors on the Flow Stack are popped and their `send_exception()` interception points are called.

This interception point may also raise a `ForwardRequest` exception (see [“ForwardRequest” on page 222](#) for more information). If an Interceptor raises this exception, no other Interceptors' `receive_request_service_contexts()` methods are called. Those Interceptors on the Flow Stack are popped and their `send_other` interception points are called.

Parameter	Description
<code>_ri</code>	This is the <code>ServerRequestInfo</code> instance to be used by Interceptor.

```
virtual void receive_request(ServerRequestInfo_ptr _ri) = 0;
```

This `receive_request()` interception point allows an Interceptor to query request information after all the information, including method parameters, are available. This interception point will execute in the same thread as the target invocation.

In the DSI model, since the parameters are first available when the user code calls `arguments()`, `receive_request()` is called from within `arguments()`. It is possible that `arguments()` is not called in the DSI model. The target may call `set_exception()` before calling `arguments()`. The ORB will guarantee that `receive_request()` is called once, either through `arguments()` or through `set_exception()`. If it is called through `set_exception()`, requesting the `arguments()` will result in `NO_RESOURCES` being raised with a standard minor code of 1.

This interception point may raise a system exception. If it does, no other Interceptors' `receive_request()` methods are called. Those Interceptors on the Flow Stack are popped and their `send_exception` interception points are called.

This interception point may also raise a `ForwardRequest` exception (see [“ForwardRequest” on page 222](#) for more information). If an Interceptor raises this exception, no other Interceptors' `receive_request()` methods are called. Those Interceptors on the Flow Stack are popped and their `send_other()` interception points are called.

Parameter	Description
<code>_ri</code>	This is the <code>ServerRequestInfo</code> instance to be used by Interceptor.

```
virtual void send_reply(ServerRequestInfo_ptr _ri) = 0;
```

This `send_reply()` interception point allows an Interceptor to query reply information and modify the reply service context after the target operation has been invoked and before the reply is returned to the client. This interception point will execute in the same thread as the target invocation.

This interception point may raise a system exception. If it does, no other Interceptors' `send_reply()` interception points are called. The remaining Interceptors in the Flow Stack will have their `send_exception()` interception point called.

Parameter	Description
<code>_ri</code>	This is the <code>ServerRequestInfo</code> instance to be used by Interceptor.

```
virtual void send_exception(ServerRequestInfo_ptr _ri) = 0;
```

This `send_exception()` interception point is called when an exception occurs. It allows an Interceptor to query the exception information and modify the reply service context before the exception is raised to the client. This interception point will execute in the same thread as the target invocation.

This interception point may raise a system exception. This has the effect of changing the exception which successive Interceptors popped from the Flow Stack receive on their calls to `send_exception`. The exception raised to the client will be the last exception raised by an Interceptor, or the original exception if no Interceptor changes the exception.

This interception point may also raise a `ForwardRequest` exception (see [“ForwardRequest” on page 222](#) for more information). If an Interceptor raises this exception, no other Interceptors' `send_exception()` interception points are called. The remaining Interceptors in the Flow Stack will have their `send_other` interception points called.

```
virtual void send_other(ServerRequestInfo_ptr _ri) = 0;
```

This `send_other()` interception point allows an Interceptor to query the information available when a request results in something other than a normal reply or an exception. For example, a request could result in a retry (e.g., a GIOP Reply with a `LOCATION_FORWARD` status was received). This interception point will execute in the same thread as the target invocation.

This interception point may raise a system exception. If it does, no other Interceptors' `send_other()` methods are called. The remaining Interceptors in the Flow Stack will have their `send_exception` interception points called.

This interception point may also raise a `ForwardRequest` exception.

Chapter 13

5.x Interceptor and object wrapper interfaces and classes

This section describes the interfaces and classes that you can use with 5.x *interceptors* and *object wrappers*.

For more information, see “Using VisiBroker Interceptors” and “Using Object Wrappers” in the *VisiBroker for C++ Developer's Guide*.

Introduction

5.x Interceptors are interceptors that are defined and implemented in VisiBroker version 5.x. Similar to Portable Interceptor, 5.x interceptors offer the VisiBroker ORB services a mechanism to intercept normal flow of execution of the ORB. The table below lists the three forms of 5.x interceptor.

Table 13.1 Types of Interceptor

Interceptor Type	Description
Client Interceptor	System level interceptors which can be used to hook ORB services such as transactions and security into the client ORB processing.
Server Interceptor	System level interceptors which can be used to hook ORB services such as transactions and security into the server ORB processing.
Object Wrappers	User level interceptors which provide a simple mechanism for users to intercept calls to stubs and skeletons. These allow for simple tracing and data caching among other things.

InterceptorManagers

Interceptors are installed and managed via interceptor managers. The `InterceptorManager` interface is the generic interceptor manager from which all interceptor-specific managers inherit. An `InterceptorManager` type is associated with each interceptor type. An `InterceptorManager` holds a list or chain of a particular kind of interceptors, all of which have the same scope and need to start at the same time. Therefore, global interceptors, such as `POALifeCycle` and `Bind` have global `InterceptorManagers` while scoped interceptors, per-POA and per-object, have an `InterceptorManager` for each scope. Each scope, either global, POAs, or objects, may hold multiple types of interceptors. You get the right kind of manager for a particular interceptor from an `InterceptorManagerControl`.

Global interceptors may be handed additional interceptor managers to install localized interceptors, for example, per-POA interceptors use the `POAInterceptorManager`.

To obtain an instance of the global interceptor manager, `InterceptorManager`, call `ORB.resolve_initial_references` and pass the `String` `InterceptorManager` as an argument. This value is only available when the ORB is in administrative mode, that is, during ORB initialization. It can only be used to install global interceptors such as, `POALifeCycle` interceptors or `Bind` interceptors.

The POA interceptor manager is a per-POA manager and is only available to `POALifeCycleInterceptors` during their `create` call. `POALifeCycleInterceptors` may set up all other server side interceptors during the call to `create`. The `Bind` Interceptor Manager is a per-object manager and is only available to `Bind` interceptors during their `bind_succeeded()` call. `Bind` interceptors may set up `ClientRequest` interceptors during the `bind_succeeded` call.

IOR templates

In addition to the interceptor, the Interoperable Object Reference (IOR) template may be modified directly on the `POAIntercptorManager` interface during the call to `POALifeCycleInterceptor::create()`. The IOR template is a full IOR value with the `type_id` not set, and all `GIOP::ProfileBodyValues` have incomplete object keys. The POA sets the `type_id` and fills in the object keys of the template before calling the `IORCreationInterceptors`.

InterceptorManager

```
class Interceptor::InterceptorManager
```

This is the base class from which all interceptor managers are derived. Interceptor managers are interfaces which are used to manage the installation and removal of interceptors from the system.

InterceptorManagerControl

```
class Interceptor::InterceptorManagerControl public CORBA::PseudoObject
```

This is the class that is responsible for controlling a set of related interceptor managers. It holds all available managers identified by a string that corresponds to the type of interceptors to be managed. There is one `InterceptorManagerControl` per scope.

Include file

Include the `interceptor_c.hh` file when you use this class.

InterceptorManagerInterceptor method

```
InterceptorManager_ptr get_manager(const char name);
```

This method returns an instance of the `InterceptorManager` which returns a string identifying the manager.

Parameter	Description
<code>name</code>	The name of the interceptor.

BindInterceptor

```
class Interceptor::BindInterceptor public VISPPseudoInterface
```

You can use this class to derive your own interceptor for handling bind and rebind events for a client or server application. The Bind Interceptors are global interceptors invoked on the client side before and after binds.

If an exception is thrown during a bind, the remaining interceptors in the chain are not called and the chain is truncated to only those interceptors already called. Exceptions thrown during `bind_succeeded` or `bind_failed` are ignored.

Include file

You should include the `interceptor_c.hh` file when you use this class.

BindInterceptor methods

```
virtual IOP::IORValue_ptr bind(IOP::IORValue_ptr ior, CORBA::Object_ptr objCORBA::Boolean rebind, VISPClosure& closure);
```

This method is called during all ORB bind operations.

Parameter	Description
<code>ior</code>	The Interoperable Object Reference (IOR) for the server object to which the client is binding.
<code>obj</code>	The client object which is being bound to the server. The object will not be properly initialized at this time, so do not attempt an operation on it. However, it may be stored in a data structure and used after the bind has completed.
<code>rebind</code>	An attempt to rebind to the server. After a <code>bind()</code> has failed, depending on the current quality of service, a rebind may be attempted.
<code>closure</code>	A new closure object for the bind operation. The closure will be used in corresponding calls to either <code>bind_failure</code> or <code>bind_succeeded</code> .
<code>return</code>	Returns a new IOR, if the bind operation is to be continued using this new IOR. Otherwise, it returns a null value and the bind will proceed using the original IOR. Returning the same IOR as the parameter passed in is incorrect and generates an exception at bind time.

```
virtual IOP::IORValue_ptr bind_failed(IOP::IORValue_ptr ior, CORBA::Object_ptr obj, VISClosure& closure);
```

This method is called if a bind operation failed.

Parameter	Description
<code>ior</code>	The IOR of the server object on which the bind operation failed.
<code>obj</code>	The client object which is being bound to the server.
<code>closure</code>	The closure object previously given in the bind call.
<code>return</code>	Returns a new IOR if a rebind is to be attempted against this IOR. Otherwise, it returns null, and a rebind is not attempted.

```
virtual void bind_succeeded(IOP::IORValue_ptr ior, CORBA::Object_ptr obj, CORBA::Long profileIndex, InterceptorManagerControl_ptr interceptorControl, VISClosure& closure);
```

This method is called if a bind operation succeeded.

Parameter	Description
<code>ior</code>	The IOR of the server object on which the bind operation succeeded.
<code>obj</code>	The client object which is being bound to the server.
<code>profileIndex</code>	Identifies the connection protocol.
<code>interceptorControl</code>	This Manager provides a list of the types of Managers.
<code>closure</code>	The closure object previously given in the bind call.

BindInterceptorManager

```
class Interceptor::BindInterceptorManager public InterceptorManager, public VISPseudoInterface
```

This is the class that manages all the global bind interceptors. It only has one public method, which allows you to register interceptors.

The `BindInterceptorManager` must always be used at `ORB_init()`. It has no effect after the orb is initialized. Therefore, it only needs to be used in the context of a loader class that inherits from `VISinit`.

To obtain a `BindInterceptorManager` from the `InterceptorManagerControl`, use `InterceptorManagerControl::get_manager()` with the identification string `Bind`.

Include file

You should include the `interceptor_c.hh` file when you use this class.

BindInterceptorManager method

```
void add (BindInterceptor_ptr interceptor)
```

This method is used to add a `BindInterceptor` to the list of interceptors to be started at bind time.

ClientRequestInterceptor

```
class Interceptor::ClientRequestInterceptor public VISPpseudoInterface
```

You use this class to derive your own client side interceptor. The Client Request interceptors may be installed during the `bind_succeeded` call of a bind interceptor and remain active for the duration of the connection. The methods defined in your derived class will be invoked by the ORB during the preparation or sending of an operation request, during the receipt of a reply message, or if an exception is raised.

Include file

Include the `interceptor_c.hh` file when you use this class.

ClientRequestInterceptor methods

```
virtual void preinvoke_premarshal (CORBA::Object_ptr target, const char*
operation, IOP::ServiceContextList& service_contexts, VisClosure& closure);
```

This method is invoked by the ORB on every request, before the request has been marshalled. An exception thrown from this interceptor results in the request being completed immediately. In this case, the chain is shortened to only those interceptors that have already fired, the request will not be sent, and `exception_occurred()` is called on all interceptors still in the chain.

Parameter	Description
<code>target</code>	The client object which is being bound to the server.
<code>operation</code>	The name of the operation being invoked.
<code>service_context</code>	The services assigned by the ORB. These services are identified by a tag registered with the OMG.
<code>closure</code>	The closure object previously given in the bind call.

```
virtual void preinvoke_postmarshal (CORBA::Object_ptr target,
CORBA_MarshallOutBuffet& payload, VISClosure& closure);
```

This method is invoked after every request has been marshaled, but before it was sent.

If an exception is thrown in this method:

- the rest of the chain is not invoked,
- the request is not sent to the server, and
- `exception_occurred()` is called on the whole interceptor chain.

Parameter	Description
<code>target</code>	The client object which is being bound to the server.
<code>payload</code>	Marshaled buffer.
<code>closure</code>	The closure object previously given in the bind call.

```
virtual void postinvoke(CORBA::Object_ptr target, const IOP::ServiceContextList&
Service_contexts, CORBA_MarshallInBuffer& payload, CORBA::Environment_ptr env,
VISClosure& closure);
```

This method is invoked after a request completes correctly or by throwing an exception. It is called after the `ServantLocator` has been invoked. Should an interceptor in the chain throw an exception, that interceptor also calls `exceptionoccurred()` and all remaining interceptors in the chain call `exception()` instead of calling `postinvoke()`.

The `CORBA::Environment` parameter is changed to reflect this exception, even when a two-way call had already written an exception in that argument.

Parameter	Description
<code>target</code>	The client object which is being bound to the server.
<code>service_context</code>	The client object which is being bound to the server. Service context length is 0 for one-way calls and during exception.
<code>payload</code>	Marshaled buffer.
<code>env</code>	Contains information on the exception that was raised.
<code>closure</code>	The closure object previously given in the bind call.

```
virtual void exception_occurred(CORBA::Object_ptr target, CORBA::Environment_ptr
env, VISClosure& closure);
```

This method is invoked by the ORB when an exception is thrown before the invocation. All exceptions thrown after the invocation are gathered in the environment parameter of the `postinvoke` method.

Parameter	Description
<code>target</code>	The client object which is being bound to the server.
<code>env</code>	Contains information on the exception that was raised.
<code>closure</code>	The closure object previously given in the bind call.

ClientRequestInterceptorManager

```
class Interceptor::ClientRequestInterceptorManager:public InterceptorManager,
public VISpseudoInterface
```

This is the class that holds the chain of `ClientRequestInterceptors` for the current object.

A `ClientRequestInterceptorManager` should be used inside of the `BindInterceptor::bind_succeeded()` method within the scope set by the `InteceptorManagerControl` passed as an argument to `bind_succeeded()`.

Include file

Include the `interceptor_c.hh` when you use this class.

ClientRequestInterceptorManager methods

```
virtual void add (ClientRequestInterceptor_ptr interceptor);
```

This method may be invoked to add a `ClientRequestInterceptor` to the local chain.

```
virtual void remove (ClientRequestInterceptor_ptr interceptor);
```

This method removes a `ClientRequestInterceptorManager`.

POALifeCycle Interceptor

```
class InterceptorManager::POALifeCycleletInterceptor public VISPpseudoInterface
```

The `POALifeCycleInterceptor` is a global interceptor which is invoked every time a POA is created or destroyed. All other server side interceptors may be installed either as global interceptors or for specific POAs. You install the `POALifeCycleInterceptor` through the `POALifeCycleInterceptorManager` interface. See [“POALifeCycleInterceptorManager” on page 248](#) for more information. The `POALifeCycleInterceptor` is called during POA creation and destruction.

Include file

Include the `PortableServerExt_c.hh` file when you use this class.

POALifeCycleInterceptor methods

```
virtual void create(PortableServer::POA_ptr poa, CORBA::PolicyList&
policiesIOP::IORValue*& iorTemplate, interceptor::InterceptorManagerControl_ptr
poaAdmin);
```

This method is invoked when a new POA is created either explicitly through a call to `create_POA` or via `AdapterActivator`. With `AdapterActivator`, the interceptor is called only after the `unknown_adapter` method successfully returns from the `AdapterActivator`. The `create` method is passed as a reference to the recently created POA and as a reference to that POA instance's `POAInterceptorManager`.

Parameter	Description
<code>poa</code>	The ID associated with the current POA being created.
<code>policies</code>	The policies for the POA being created.
<code>iorTemplate</code>	The IOR template is a full IOR value with the <code>type_id</code> not set, and all <code>GIOP::ProfileBodyValues</code> will have incomplete object keys.
<code>poaAdmin</code>	The control for the POA being created. See “InterceptorManagerControl” on page 242 for more information.

```
virtual void destroy(PortalServer::POA_ptr poa);
```

This method is called before a POA is destroyed and all of its objects have been etherialized. It guarantees that `destroy` will be called on all interceptors before `create` will be called again for a POA with the same name. If the `destroy` operation throws a system exception, the exception is ignored, and the remaining interceptors are called.

Parameter	Description
<code>poa</code>	Portable Object Adapter (POA) being destroyed.

POALifeCycleInterceptorManager

```
class InterceptorExt::POALifeCycleInterceptorManager public
  interceptor::InterceptorManager, public VISPpseudoInterface
```

This class manages all POALifeCycle global interceptors. There is a single instance of the POALifeCycleInterceptorManager defined in an ORB.

Then scope of this interface is global, per-ORB. This class is only active during ORB_init() time.

Include file

Include the **PortalServerExt_c.hh** file when you use this class.

POALifeCycleInterceptorManager method

```
virtual void add(POALifeCycleInterceptor_ptr interceptor);
```

This method may be invoked to add a POALifeCycleInterceptor to the global chain of POALifeCycle interceptors.

Parameter	Description
interceptor	The interceptor to be added.

ActiveObjectLifeCycleInterceptor

```
class PortableServerExt::ActiveObjectLifeCycleInterceptor public
  VISPpseudoInterface
```

The ActiveObjectLifeCycleInterceptor interceptor is called when objects are added and removed from the active object map. Only used when POA has RETAIN policy. This class is a POA-scoped interceptor which may be installed by a POALifeCycleInterceptor when the POA is created.

Include file

Include the **PortableServerExt_c.hh** file when you use this class.

ActiveObjectLifeCycleInterceptor methods

```
virtual void create(const PortableServer::ObjectId& oid,
  PortableServer::ServantBase* servant, PortableServer::POA_ptr adapter);
```

This method is invoked after an object has been added to the Active Object Map, either through explicit or implicit activation, using either direct APIs or a ServantActivator. The object reference and the POA of the new active object are passed as parameters.

Parameter	Description
oid	Object ID for the object currently activated.
servant	Associated servant
adapter	The Portable Object Adapter (POA) being created or destroyed.


```
virtual void destroy(const PortableServer::ObjectId& oid,
PortableServer::ServantBase* servant, PortableServer::POA_ptr adapter);
```

This method is called after an object has been deactivated and etherialized. The object reference and the POA of the object are passed as parameters.

Parameter	Description
oid	Object ID for the object currently activated
servant	Associated servant
adapter	The Portable Object Adapter (POA) being created or destroyed.

ActiveObjectLifeCycleInterceptorManager

```
class PortableServerExt::ActiveObjectLifeCycleInterceptorManager public
interceptor::InterceptorManager, public VISPPseudoInterface
```

This is the class that manages all `ActiveObjectLifeCycleInterceptors` registered in its scope. Each POA has one single `ActiveObjectLifeCycleInterceptorManager`.

Include file

Include the `PortableServer_c.hh` file when you use this class.

ActiveObjectLifeCycleInterceptorManager method

```
virtual void add(ActiveObjectLifeCycleInterceptor interceptor_ptr interceptor);
```

This method may be invoked to add an `ActiveObjectLifeCycleInterceptor` to the chain.

ServerRequestInterceptor

```
class Interceptor::ServerRequestInterceptor public VISPPseudoInterface
```

The `ServerRequestInterceptor` class is a POA-scoped interceptor which may be installed by a `POALifeCycleInterceptor` at POA creation time. This class may be used to perform access control, to examine and insert service contexts, and to change the reply status of a request.

Include file

Include the `interceptor_c.hh` file when you use this class.

ServerRequestInterceptor methods

```
virtual void preinvoke(CORBA::Object_ptr _target, const char* operation, const
IOP::ServiceContextList& service_contexts, CORBA::MarshalInBuffer& payload,
VISClosure& closure) raises (ForwardRequestException);
```

This method is invoked by the ORB on every request, before the request is demarshaled. An exception thrown from this interceptor results in the request being completed immediately. This method is called before any `ServantLocators` are invoked. The result may be that the servant may not be available while this method is running.

Parameter	Description
target	The client object which is being bound to the server.
operation	Identifies the name of the operation being invoked.
service_contexts	Identifies the services assigned by the Orb. These services are registered with the OMG.
payload	Marshaled buffer.
closure	May contain data saved by one interceptor method that can be retrieved later by another interceptor method.

```
virtual void postinvoke_premarshal(CORBA::Object_ptr target,
IOP::ServiceContextList& ServiceContextList, CORBA::Environment_ptr env,
VISClosure& closure);
```

This method is invoked after an upcall to the servant but before marshalling the reply. An exception here is handled by interrupting the chain: the request is not sent to the server and `exceptionoccurred()` is called on all interceptors in the chain

Parameter	Description
target	The client object which is being bound to the server.
ServiceContextList	Identifies the services assigned by the Orb. These services are registered with the OMG.
env	Contains information on the exception that was raised.
closure	May contain data saved by one interceptor method that can be retrieved later by another interceptor method.

```
virtual void postinvoke_postmarshal(CORBA::Object_ptr _target,
CORBA::MarshalOutBuffer& _payload, VISClosure& _closure);
```

This method is invoked after marshalling the reply but before sending the reply to the client. Exceptions thrown here are ignored. The entire chain is guaranteed to be called.

Parameter	Description
target	The object to which that application was attempting to bind.
payload	Marshaled buffer.
closure	May contain data saved by one interceptor method that can be retrieved later by another interceptor method.

```
virtual void exception_occurred(CORBA::Object_ptr _target, CORBA::Environment_ptr
_env, VISPClosure& _closure);
```

This method is invoked by the ORB when an `exceptionoccurred` interceptor is called on all remaining interceptors in the chain after an exception occurred in one of the `prepare_reply` interceptors. An exception thrown during this call replaces the existing exception in the environment

Parameter	Description
<code>target</code>	The client object which is being bound to the server.
<code>env</code>	Contains information on the exception that was raised.
<code>closure</code>	May contain data saved by one interceptor method that can be retrieved later by another interceptor method.

ServerRequestInterceptorManager

```
class Interceptor::ServerRequestInterceptorManager public InterceptorManager,
public VISPPseudoInterface
```

This is the class that manages all `ServerRequestInterceptors` registered in its scope. Each POA has one single `ServerRequestInterceptorManager`.

Include file

Include the `interceptor_c.hh` file when you use this class.

ServerRequestInterceptorManager method

```
virtual void add(ServerRequestInterceptor_ptr interceptor);
```

Invoke this method to add a `ServerRequestInterceptor` to the chain.

IORCreationInterceptor

```
class PortableServerExt::IORInterceptor public VISPPseudoInterface
```

The `IORCreationInterceptor` is a per-POA interceptor which may be installed by a `POALifeCycleInterceptor` at POA creation time. The interceptor may be used to modify IORs by adding additional profiles or components. This class is typically used to support services such as transactions or firewall.

This kind of interceptor is used to automatically change the IOR templates on certain classes of POAs whose names and identities may not be known at development time. This may be the case with services such as Transaction and Firewall.

Note To change all the IORs created by a POA, simply modify the `IORTemplate` for that POA. The change will apply only to newly created IORs and not to any existing ones.

Making radical changes to the IOR is not recommended.

Include file

Include the `PortableServerExt_c.hh` file when you use this class.

IORInterceptor method

```
virtual void create(PortableServer::POA poa, IOP::IORValue*& ior);
```

The method is called whenever the POA needs to create an object reference. It takes the POA and the IORValue for the reference as arguments. The interceptor may modify the IORValue by adding additional profiles or components, or changing the existing profiles or components.

Parameter	Description
poa	The ID associated with the current PAO being created.
ior	The IOR for the server object to which the client is binding.

IORCreationInterceptorManager

```
class PortableServerExt::IORCreationInterceptorManager public
interceptor::InterceptorManager, public VISPPseudoInterface
```

This is the class that is used to manage (add) IOR interceptors to the local chain. Each POA has one single IORInterceptorManager.

Include file

Include the **PortableServerExt_c.hh** file when you use this class.

IORCreationInterceptorManager method

```
virtual void add(IORCreationInterceptor_ptr _interceptor);
```

This method may be invoked to add an IORInterceptor to the local chain.

Closure

```
public interface Closure extends Object
```

Closure objects are created by the ORB at the beginning of certain sequences of interceptor calls. The same Closure object is used for all calls in that particular sequence. The Closure object contains a single public data field, object, of type `java.lang.Object` which may be set by the interceptor to keep state information. The sequences for which Closure objects are created vary depending on the interceptor type.

This code sample shows the closure class.

```
class Closure {
    java.lang.Object object;
};
```

ExtendedClosure

```
public interface ExtendedClosure extends Closure {
    public RequestInfo reqInfo;
    public InputStream payload;
}
```

This interface is a derived class of `Closure` and contains a `RequestInfo` for read only attribute.

This code sample shows the `RequestInfo` IDL.

```
struct RequestInfo {
    boolean response_expected;
    unsigned long request_id;
};
```

You can cast the `Closure` object passed to the `ServerRequestInterceptor` and `ClientRequestInterceptor` to its subclass, `ExtendedClosure`. `ExtendedClosure` can be used to extract the `RequestInfo`, from which you can extract the `request_id` and `response_expected`. The `request_id` is the unique id assigned to the request. The `response_expected` flag indicates whether the request is a one-way call.

```
int my_response_expected =
    ((ExtendedClosure)closure).reqInfo.response_expected;
int my_request_id = ((ExtendedClosure)closure).reqInfo.request_id;
```

For more information, please see the example in `examples/interceptor/client_server`.

VISClosure

```
struct VISClosure
```

This structure is used to store data so that it can be shared between different invocations of interceptor methods. The data that is stored is un-typed and can represent state information related to an operation request or a bind or locate request. It is used in conjunction with the `VISClosureData` class.

Include file

Include the `vinter.h` file when you use this class.

VISClosure members

```
CORBA::ULong id
```

You can use this data member to uniquely identify this object if you are using more than one `VISClosure` object.

```
void *data
```

This data member points to the un-typed data that may be stored or accessed by an interceptor method.

```
VISClosureData *managedData
```

This data member points to the `VISClosureData` class that represents the actual data. You may cast your managed data to this type.

VISClosureData

```
class VISClosureData
```

This class represents managed data that can be shared between different invocations of interceptor methods.

VISClosureData methods

```
virtual void _VisClosureData();
```

This is the default destructor.

```
virtual void _release();
```

Releases this object and decrements the reference count. When the reference count reaches 0, the object is deleted.

ChainUntypedObjectWrapperFactory

```
class VISObjectWrapper::ChainUntypedObjectWrapperFactory:public
UntypedObjectWrapperFactory
```

This interface is used by a client or server application to add or remove an `UntypedObjectWrapperFactory` object. An `UntypedObjectWrapperFactory` is used to create an `UntypedObjectWrapper` for each object a client application binds to or for each object implementation created by a server application.

Refer to using object wrappers section in the Borland VisiBroker *Developer's Guide* for more information about how to use the object wrappers. ##Same problem as above. See above comments.##

Include file

Include the `vobjwrap.h` file when you use this class.

ChainUntypedObjectWrapperFactory methods

```
void add(UntypedObjectWrapperFactory_ptr factory,Location loc);
```

This method adds the specified un-typed object wrapper factory for a client application, server application, or collocated application.

If your application is acting as both a client application and a server application, that is, a collocated application, you can install an un-typed object wrapper factory. If you do so, the wrapper's methods are invoked for both invocations on bound objects and operation requests received by object implementations. In other words, they are invoked on both the client and server portions of the application.

Note On the client side, un-typed object wrapper factories must be defined before any objects are bound. On the server side, un-typed object wrapper factories must be defined before an invocation for an object implementation is received.

Parameter	Description
factory	A pointer to the factory to be registered.
loc	The location of the factory being added, which should be one of the following values: VISObjectWrapper::Client VISObjectWrapper::Server VISObjectWrapper::Both

```
void remove(UntypedObjectWrapperFactory_ptr factory, Location loc);
```

This method removes the specified un-typed object wrapper factory from the specified location.

If your application is acting as both a client and a server, you can remove the object wrapper factories for either the client side objects, server side implementations, or both.

Note Removing one or more object wrapper factories from a client does not affect objects of that class which are already bound by the client. Only subsequently bound objects will be affected.

Removing object wrapper factories from a server does not affect object implementations that have already serviced requests. Only subsequently created object implementations will be affected.

Parameter	Description
factory	A pointer to the factory to be registered.
loc	The location of the factory being removed; one of the following values: VISObjectWrapper::Client VISObjectWrapper::Server VISObjectWrapper::Both

```
static CORBA::ULong count(Location loc);
```

This static method returns the number of un-typed object wrapper factories installed for the specified location.

Parameter	Description
loc	The location of the factories: VISObjectWrapper::Client VISObjectWrapper::Server VISObjectWrapper::Both

UntypedObjectWrapper

```
class VISObjectWrapper::UntypedObjectWrapper : public VISResource
```

You use this class to derive and implement an un-typed object wrapper for a client application, a server application, or co-located application. When you derive an un-typed object wrapper from this class, you define a `pre_method` method that is invoked before a request is issued by a client application or before it is processed by an object implementation on the server side. You also define a `post_method` method that will be invoked after an operation request is processed by an object implementation on the server side or after a reply has been received by a client application.

You must also derive a factory class that will create your un-typed wrapper objects. Derive it from the `UntypedObjectWrapperFactory` class, described in the “[UntypedObjectWrapperFactory](#)” on page 257.

Refer to the Borland VisiBroker *Developer's Guide* for more information about how to use the object wrappers. ##See above comments.##

Include file

Include the `vobjwrap.h` file when you use this class.

UntypedObjectWrapper methods

```
virtual void pre_method(const char* operation, CORBA::Object_ptr target,
VISClosure& closure);
```

This method is invoked before an operation request is sent on the client side or before it is processed by an object implementation on the server side.

Parameter	Description
<code>operation</code>	The name of the operation being requested.
<code>target</code>	The object that is the target of the request.
<code>closure</code>	The <code>Closure</code> object can be used to pass data between object wrapper methods.

```
virtual void post_method(const char* operation, CORBA::Object_ptr target,
CORBA::Environment& env, VISClosure& closure);
```

This method is invoked after an operation request has been processed by the object implementation on the server side or before the reply message is processed by the stub on the client side.

Parameter	Description
<code>operation</code>	The name of the operation being requested.
<code>target</code>	The object that is the target of the request.
<code>env</code>	An <code>Environment</code> object that is used to reflect exceptions that might have occurred in the processing of the operation request.
<code>closure</code>	The <code>Closure</code> object can be used to pass data between object wrapper methods.

UntypedObjectWrapperFactory

```
class VISObjectWrapper::UntypedObjectWrapperFactory
```

You use this interface to derive your own un-typed object wrapper factories. Your factory will be used to create an instance of your un-typed object wrapper for an application whenever a new object is bound to or an object implementation services a request.

Include file

Include the `vobjwrap.h` file when you use this class.

UntypedObjectWrapperFactory constructor

```
UntypedObjectWrapperFactory(Location loc, CORBA::Boolean doAdd=1);
```

Creates an un-typed object wrapper factory for the specified location and by default registers it with the `ChainUntypedObjectWrapperFactory`. If your application is acting as both a client application and a server application, you can install an un-typed object wrapper factory so the wrapper's methods will be invoked for both invocations on bound objects and operation requests received by object implementations.

If you don't want to use the default parameter, you can specify that the `doAdd` not be performed. However, to create an untyped object wrapper, you will have to call `ChainUntypedObjectWrapper::add`.

Parameter	Description
<code>loc</code>	The location of the factory being added; one of the following values: <code>VISObjectWrapper::Client</code> <code>VISObjectWrapper::Server</code> <code>VISObjectWrapper::Both</code>
<code>doAdd</code>	A flag specifying whether or not the factory is to be registered.

UntypedObjectWrapperFactory methods

```
virtual UntypedObjectWrapper_ptr create(CORBA::Object_ptr target, Location loc);
```

This method is called to create an instance of your type of `UntypedObjectWrapper`. Your implementation of this method can examine the type of bound object or object implementation to determine whether or not it wants to create an object wrapper for that object. With the `loc` parameter, you specify whether the `create` request is called to wrap a client object or a server implementation.

Parameter	Description
<code>target</code>	The object being bound by a client application for which the un-typed object wrapper is being created. If this method is being invoked on the server side, this represents the object implementation that is being created.
<code>loc</code>	The location of the factory being added.

Chapter 14

Quality of Service interfaces and classes

This section describes the VisiBroker for C++ implementation of the Quality of Service APIs. See the Core interfaces and classes, “[PortableServer::POA](#)” on page 30, for information about creating policies.

CORBA::PolicyManager

```
class CORBA::PolicyManager
```

This class is used to set and access policy overrides at the VisiBroker ORB level. Policies defined at the VisiBroker ORB level override any system defaults. The instance belonging to the manager thread is accessible by using `resolve_initial_reference("PolicyManager")` and narrowing down to `PolicyManager`.

IDL definition

```
module CORBA {
    interface PolicyManager {
        PolicyList get_policy_overrides(in PolicyTypeSeq ts);
        void set_policy_overrides(in PolicyList policies, in
            SetOverrideType set_add)
            raises (InvalidPolicies);
    };
};
```

Methods

```
PolicyList get_policy_overrides (PolicyTypeSeq ts);
```

This method returns a policy list containing the policies of the requested policy types. If the specified sequence is empty (that is, if the length of the list is zero), all policies at this scope are returned. If none of the requested policy types is set at the target **PolicyManager**, an empty sequence is returned.

```
void set_policy_overrides (PolicyList policies, CORBA::SetOverrideType set_add)
```

This method updates the current set of policies with the requested list of policy overrides. To remove all overrides from a **PolicyManager**, invoke `set_policy_overrides` with an empty sequence of policies and a mode of `SET_OVERRIDE`.

Only certain policies that pertain to the invocation of an operation at the client end can be overridden using this operation. An attempt to override any other policy results in the raising of the `CORBA::NO_PERMISSION` exception. If the request would put the set of overriding policies for the target **PolicyManager** in an inconsistent state, no policies are changed or added, and the exception `InvalidPolicies` is raised. There is no evaluation of compatibility with policies set within other **PolicyManagers**.

Parameter	Description
<code>policies</code>	A sequence of references to Policy objects.
<code>set_add</code>	Indicates whether these policies should be added (<code>ADD_OVERRIDE</code>) to any other overrides that already exist in the PolicyManager or added to a clean PolicyManager free of any other overrides (<code>SET_OVERRIDE</code>).

CORBA::Object

```
class CORBA::Object
```

The **VisiBroker** implementation of the Quality of Service API allows policies to be assigned to objects, threads, and **VisiBroker** ORBs. Policies assigned to **Objects** override all other policies.

IDL definition

```
#pragma prefix "omg.org"
module CORBA {
  interface Object {
    Policy get_client_policy(in PolicyType type);
    Policy get_policy(in PolicyType type);
    PolicyList get_policy_overrides(in PolicyTypeSeq types);
    Object set_policy_overrides(in PolicyList policies,in SetOverrideType
      set_add)
      raises (InvalidPolicies);
    boolean validate_connection(out PolicyList inconsistent_policies);
  };
};
```

Methods

```
CORBA::Policy_ptr get_client_policy(CORBA::PolicyType type);
```

Returns the effective overriding Policy for the object reference. The effective override is obtained by first checking for an override of the specified `PolicyType` at the Object scope, then at the Current scope, and finally at the VisiBroker ORB scope. If no override is present for the requested `PolicyType`, the system-dependent default value for that `PolicyType` is used. Portable applications are expected to set the desired “defaults” at the VisiBroker ORB scope since default Policy values are not specified.

```
CORBA::Policy_ptr get_policy(CORBA::PolicyType type);
```

Returns the effective Policy for the object reference. The effective Policy is the one that would be used if a request were made. This Policy is determined first by obtaining the effective override for the `PolicyType` as returned by `get_client_policy`.

The effective override is then compared with the Policy as specified in the IOR. The effective Policy is the intersection of the values allowed by the effective override and the IOR-specified Policy. If the intersection is empty, the system exception `INV_POLICY` is raised. Otherwise, a Policy with a value legally within the intersection is returned as the effective Policy. The absence of a Policy value in the IOR implies that any legal value may be used. To ensure the accuracy of the returned effective Policy, invoke `non_existent` or `validate_connection` on an object reference prior to `get_policy`. If `get_policy` is invoked prior to the object reference being bound, the returned effective Policy is implementation-dependent. In that situation, a compliant implementation may do any of the following: raise the exception `CORBA::BAD_INV_ORDER`, return some value for that `PolicyType` which may be subject to change once a binding is performed, or attempt a binding and then return the effective Policy. Note that if the `RebindPolicy` has a value of `TRANSPARENT`, the effective Policy may change from invocation to invocation due to transparent rebinding.

Note In the VisiBroker implementation, this method gets the Policy assigned to an Object, thread or the VisiBroker ORB.

```
CORBA::Object set_policy_overrides(const PolicyList& _policies,
CORBA::SetOverrideType _set_add);
```

This method works as does the `PolicyManager` method of the same name. However, this method updates the current set of policies of an Object, thread or the VisiBroker ORB with the requested list of Policy overrides. In addition, this method returns a `CORBA::Object` whereas other methods of the same name return `void`.

```
CORBA::Boolean validate_connection(PolicyList inconsistent_policies);
```

Returns the value `TRUE` if the current effective policies for the Object will allow an invocation to be made. If the object reference is not yet bound, a binding will occur as part of this operation. If the object reference is already bound, but current policy overrides have changed or for any other reason the binding is no longer valid, a rebind will be attempted regardless of the setting of any `RebindPolicy` override. The `validate_connection` operation is the only way to force such a rebind when implicit rebinds are disallowed by the current effective `RebindPolicy`. The attempt to bind or rebind may involve processing GIOP `LocateRequests` by the VisiBroker ORB.

This method returns the value `FALSE` if the current effective policies would cause an invocation to raise the system exception `INV_POLICY`. If the current effective policies are incompatible, the out parameter `inconsistent_policies` contains those policies causing the incompatibility. This returned list of policies is not guaranteed to be exhaustive. If the binding fails due to some reason unrelated to policy overrides, the appropriate system exception is raised.

Messaging::RebindPolicy

```
class Messaging::RebindPolicy
```

The `VisiBroker` implementation of `RebindPolicy` is a complete implementation of `RebindPolicy` as defined in the `Messaging Specification` with enhancements to support failover.

The `RebindPolicy` of the `VisiBroker ORB` determines how it handles GIOP location-forward messages and object failures. The `VisiBroker ORB` handles fail-over/rebind by looking at the effective policy at the `CORBA::Object` instance.

The `OMG` implementation, derived from `CORBA::Policy`, determines whether the `VisiBroker ORB` may transparently rebind once it is successfully bound to a target server. The extended implementation determines whether the `VisiBroker ORB` may transparently failover once it is successfully bound to a target `Object`, `thread`, or `VisiBroker ORB`.

IDL definition

```
#pragma prefix "omg.org"
module Messaging {
    typedef short RebindMode;
    const CORBA::PolicyType REBIND_POLICY_TYPE = 23;
    interface RebindPolicy CORBA::Policy {
        readonly attribute RebindMode rebind_mode;
    };
}
```

Policy values

Note Policies are enforced only after a successful bind.

The `OMG` Policy values that can be set as the `Rebind Policy` are:

Policy Value	Description
<code>TRANSPARENT</code>	This policy allows the <code>VisiBroker ORB</code> to silently handle object-forwarding and necessary reconnection when making a remote request. This is the least restrictive <code>OMG</code> policy value.
<code>NO_REBIND</code>	This policy allows the <code>VisiBroker ORB</code> to silently handle reopening of closed connections while making a remote request, but prevents any transparent object-forwarding that would cause a change in the client-side effective <code>QoS</code> policies.
<code>NO_RECONNECT</code>	This policy prevents the <code>VisiBroker ORB</code> from silently handling object-forwards or the reopening of closed connections. This is the most restrictive <code>OMG</code> policy value.

The VisiBroker-specific values that can be set as the Rebind Policy are:

Policy Value	Description
VB_TRANSPARENT	This policy extends <code>TRANSPARENT</code> behavior to failover conditions in the object, the thread and the VisiBroker ORB. This is the default policy. If this policy is set, if a remote invocation fails because the server object goes down, then the VisiBroker ORB tries to reconnect to another server using the osagent. The VisiBroker ORB masks the communication failure and does not throw an exception to the client.
VB_NOTIFY_REBIND	<code>VB_NOTIFY_REBIND</code> behaves as does <code>VB_TRANSPARENT</code> but throws an exception when the communication failure is detected. It will try to transparently reconnect to another object if the invocation is re-attempted.
VB_NO_REBIND	<code>VB_NO_REBIND</code> does no failover. It only allows the client VisiBroker ORB to reopen a closed GIOP re-connection to the same server; it does not allow object forwarding of any kind.

QoSExt::DeferBindPolicy

```
class QoSExt::DeferBindPolicy
```

By default, the VisiBroker ORB connects to the (remote) object when it receives a `bind()` call.

If set to `TRUE`, this policy changes this behavior; it causes the VisiBroker ORB to delay contacting the Object until the first invocation.

IDL definition

```
#pragma prefix "inprise.com"
module QoSExt {
    interface DeferBindPolicy :CORBA::Policy {
        readonly attribute boolean value;
    };
};
```

QoSExt::RelativeConnectionTimeoutPolicy

```
class QoSExt::RelativeConnectionTimeoutPolicy
```

The `RelativeConnectionTimeoutPolicy` indicates a timeout after which attempts to connect to an object using one of the available endpoints is aborted. The policy value of unsigned longlong type specifies the timeout in 100s of nanoseconds. It is applied to every endpoint that the VisiBroker ORB tries to connect to. Therefore, if multiple connection attempts are made, the elapsed time will be a multiple of the configured timeout. The default value of 0 sets the timeout value to that of the operating system default timeout.

Note This policy is not enforced in local IPC or in-process communication. By default, in-process and local IPC have higher precedence than other communication, therefore you should turn off in-process and local IPC if you need to ensure this policy works. Turn off local IPC if timeout policies are required to ensure their interoperability with the Java and C++ VisiBroker ORBs or with other ORB vendors.

IDL definition

```
module QoSExt {
    const CORBA::PolicyType RELATIVE_CONN_TIMEOUT_POLICY_TYPE = 0x56495304;
    interface RelativeConnectionTimeoutPolicy :CORBA::Policy {
        readonly attribute TimeBase::TimeT relative_expiry;
    };
};
```

Messaging::RelativeRequestTimeoutPolicy

```
class Messaging::RelativeRequestTimeoutPolicy
```

The `RelativeRequestTimeoutPolicy` specifies the maximum time that a client is to block waiting to send an operation request. If the request times out, `CORBA::TIMEOUT` exception is raised and the connection to the server is destroyed.

Chapter 15

IOP and IIOP interfaces and classes

This section describes the VisiBroker for C++ implementation of the key General Inter-ORB Protocol interfaces and other structures defined by the CORBA specification. For a complete description of these interfaces, refer to the *OMG CORBA/IIOP Specification*.

GIOP::MessageHeader

```
struct MessageHeader
```

This structure is used to represent information about a GIOP message.

MessageHeader members

```
CORBA::Char magic[4]
```

This string should always contain GIOP.

```
Version GIOP_version
```

Indicates the version of the protocol being used. This structure contains a major and minor version number, as shown. The major version should be set to 1 and the minor version should be set to 2, unless you are using an older version; for example, VisiBroker 3.x, in which case the minor version should be set to 0.

```
struct Version {  
    CORBA::Octet    major;  
    CORBA::Octet    minor;  
};
```

```
CORBA::Boolean byte_order
```

Set to `TRUE` to indicate that little-endian byte ordering is used in the message. If set to `FALSE`, big-endian byte ordering is used in the message.

CORBA::Octet **message_type**

Indicates the type of message that follows the header. This should be one of the following values:

```
enum MsgType {
    Request,
    Reply,
    CancelRequest,
    LocateRequest,
    LocateReply,
    CloseConnection,
    MessageError,
    Fragment
};
```

CORBA::ULong **message_size**

Indicates the length of the message that follows this header.

GIOP::CancelRequestHeader

struct **CancelRequestHeader**

This structure is used to represent information about a cancel request message header.

CancelRequestHeader members

CORBA::ULong **request_id**

This data member represents the request identifier that is being canceled.

GIOP::LocateReplyHeader

struct **LocateReplyHeader**

This structure is used to represent a message that is sent in reply to a locate request message. Additional data follows this header if the `locate_status` is set to `OBJECT_FORWARD`.

LocateReplyHeader members

CORBA::ULong **request_id**

The request identifier of the original request.

LocateStatusType **locate_status**

Indicates the disposition of the locate request as one of the following values:

Value	Description
UNKNOWN_OBJECT	Indicates the requested object could not be found. No other data is associated with this message.
OBJECT_HERE	Indicates the object is implemented by this server. No other data is associated with this message.

Value	Description
OBJECT_FORWARD	Indicates that the reply contains an object reference (IOR) that may be used as the target for requests to the object specified in the LocateRequest message. The object is implemented by another server and a IOR for that server follows this header.
OBJECT_FORWARD_PERM	Indicates that the reply contains an object reference (IOR) that may be used as the target for requests to the object specified in the LocateRequest message.
LOC_SYSTEM_EXCEPTION	Indicates that the exception contains a marshaled GIOP::System ExceptionReplyBody.
LOC_NEEDS_ADDRESSING_MODE	Indicates that the requested addressing mode will be used when the LocateRequest is re-sent.

GIOP::LocateRequestHeader

```
structure LocateRequestHeader
```

This structure represents a message containing a request to locate an object.

LocateRequestHeader members

```
CORBA::ULong request_id
```

The request identifier for this message. It is used to distinguish between multiple outstanding messages.

```
GIOP::TargetAddress target
```

The object to be located. The target is a union of three different things: object key, profile, and IOR.

GIOP::ReplyHeader

```
struct ReplyHeader
```

This structure represents the reply header of a reply message that is sent to a client in response to a request message.

Include file

The **GIOP_c.hh** file should be included when you use this structure. This file is already included in corba.h in the installation/include directory.

ReplyHeader members

```
CORBA::ULong request_id
```

Should be set to the same `request_id` as the request message with which this reply is associated.

ReplyStatusType **reply_status**

Indicates the status of the reply and should be set to one of the following enum values:

- NO_EXCEPTION
- USER_EXCEPTION
- SYSTEM_EXCEPTION
- LOCATION_FORWARD
- LOCATION_FORWARD_PERM
- NEEDS_ADDRESSING_MODE

IOP::ServiceContextList **service_info**

A list of service context information that may be passed from the server to the client.

GIOP::RequestHeader

struct **RequestHeader**

This structure represents the request header of a request message that is sent to an object implementation.

Include file

The **GIOP_c.hh** file should be included when you use this structure. This file is already included in corba.h in the installation/include directory.

RequestHeader members

CORBA::ULong **request_id**

A unique identifier used to associate a reply message with a particular request message.

CORBA::Boolean **response_expected**

This member is **FALSE** if the request is a oneway operation for which a reply is not expected. This member is **TRUE** for operation requests and other requests that expect a reply.

GIOP::TargetAddress **_target**

The object that is the target of the request. The target is a union of the following: object key, profile, and IOR. Object keys are stored in a vendor-specific format and are generated when an IOR is created.

CORBA::String_var **oper**

Identifies the operation being requested on the target object. This member is the same as the **operator** member, except that it is a managed type.

const char ***operation**

Identifies the operation being requested on the target object. This member is the same as the **oper** member, except that it is not a managed type.

IOP::ServiceContextList **service_context**

A list of service context information that may be passed from the client to the server.

IIOP::ProfileBody

struct **ProfileBody**

This structure contains information about the protocol supported by an object.

```
module IIOP {
    :
    struct ProfileBody {
        Version iiop_version;
        string host;
        unsigned short port;
        sequence<octet> object_key;
        sequence<IOP::taggedComponent> components;
    };
};
```

ProfileBody members

Version **iiop_version**

The version of IIOP supported.

CORBA::String_var **host**

The name of the host where the server hosting the object is running.

CORBA::UShort **port**

The port number to use for establishing a connection to the server hosting the object.

CORBA::OctetSequence **object_key**

Object keys are stored in a vendor-specific format and are generated when an IOR is created.

IIOP::MultiComponentProfile **components**

A sequence of TaggedComponents which contain information about the protocols that are supported.

IOP::IOR

struct **IOR**

This structure represents an Interoperable Object Reference and is used to provide important information about object references. Your client application can create a stringified IOR by invoking the `ORB::object_to_string` method.

Include file

The `IOP_c.hh` file should be included when you use this structure.

IOR members

CORBA::String_var **type_id**

This data member describes the type of object reference that is represented by this IOR.

TaggedProfileSequence **profiles**

This data member represents a sequence of one or more `TaggedProfile` structures which contain information about the protocols that are supported.

IOP::TaggedProfile

struct **TaggedProfile**

This structure represents a particular protocol that is supported by an Interoperable Object Reference (IOR).

TaggedProfile members

ProfileID **tag**

The contents of the profile data. Its value should be one of the following:

Value	Description
TAG_INTERNET_IOP	Indicates the protocol is standard IIOP.
TAG_MULTIPLE_COMPONENTS	Indicates the profile data contains a list of VisiBroker ORB services available using the protocol.
TAG_VB_LOCATOR	Indicates that the IOR is an interim, pseudo-object that is used until the real IOR is received by the osagent.
TAG_VSGN_LIOP	Indicates the protocol is IOP over a local IPC mechanism.

CORBA_OctetSequence **profile_data**

This data member encapsulates all the protocol information needed to invoke an operation on an IOR.

Marshal buffer interfaces and classes

This section describes the buffer class used for marshalling data to a buffer when creating an operation request or a reply message. It also describes the buffer class used for extracting data from a received operation request or reply message.

CORBA::MarshalInBuffer

```
class CORBA::MarshalInBuffer : public VISistream
```

This class represents a stream buffer that allows IDL types to be read from a buffer. Interceptor methods that you implement may use this class. See [“Portable Interceptor interfaces and classes” on page 213](#) for more information on the interceptor interfaces.

The `CORBA::MarshalInBuffer` class is used on the client side to extract the data associated with a reply message. It is used on the server side to extract the data associated with an operation request. This class provides a wide range of methods for retrieving various types of data from the buffer.

This class also provides several static methods for testing and manipulating `CORBA::MarshalInBuffer` pointers.

A `CORBA::MarshalInBuffer_var` class is also offered. It provides a wrapper that automatically manages the contained object.

Include file

The `mbuf.h` file should be included when you use this class. This file gets included in `corba.h`. So, you don't have to include `mbuf.h` separately.

CORBA::MarshalInBuffer constructors/destructors

```
CORBA::MarshalInBuffer(char *read_buffer, CORBA::ULong length, CORBA::Boolean
release_flag=0, CORBA::ULong start_offset=0, CORBA::Boolean byte_order =
CORBA::ByteOrder);
```

This is the default constructor.

Parameter	Description
read_buffer	The buffer where the marshalled data will actually be stored.
length	The maximum number of bytes that may be stored in read_buffer.
release_flag	If set to TRUE, the memory associated with read_buffer is freed when this object is destroyed. The default value is FALSE.
start_offset	The starting offset wherein data is written in the read_buffer. The default value is 0.
byte_order	Set this to TRUE to indicate that little-endian byte ordering is being used. Set to FALSE to indicate that big-endian byte ordering is being used.

```
virtual ~CORBA::MarshalInBuffer();
```

This is the default destructor. The buffer memory associated with this object is released if the release_flag is set to TRUE. The release_flag may be set when the object is created or by invoking the release_flag method, described in “void release_flag(CORBA::Boolean val);” on page 274.

CORBA::MarshalInBuffer methods

```
char *buffer() const;
```

Returns a pointer to the buffer associated with this object.

```
void byte_order(CORBA::Boolean val) const;
```

Sets the byte ordering for this message buffer.

Parameter	Description
val	Set this to TRUE to indicate that little-endian byte ordering is being used. Set to FALSE to indicate that big-endian byte ordering is being used.

```
CORBA::Boolean byte_order() const;
```

Returns TRUE if the buffer uses little-endian byte ordering. FALSE is returned if big-endian byte ordering is used.

```
CORBA::ULong curoff() const;
```

Returns the current offset within the buffer associated with this object.


```
virtual VISistream& get(char& data); virtual VISistream& get(unsigned char& data);
```

These methods allow you to retrieve a single character from the buffer at the current location.

This method returns a pointer to the location within the buffer immediately following the end of the data that was just retrieved.

Parameter	Description
data	The location where the retrieved char or unsigned char is to be stored.

```
virtual VISistream& get(<data_type> data, unsigned size);
```

These methods allow you to retrieve a sequence of data from the buffer at the current location. There is a separate method for each of the listed target data types.

This method returns a pointer to the location within the buffer immediately following the end of the data that was just retrieved.

Parameter	Description														
data	The location where the retrieved data is to be stored. The supported target data types are: <table border="0" style="margin-left: 20px;"> <tr> <td>char*</td> <td>unsigned long*</td> </tr> <tr> <td>unsigned char*</td> <td>float*</td> </tr> <tr> <td>short*</td> <td>double*</td> </tr> <tr> <td>unsigned short*</td> <td>long double*</td> </tr> <tr> <td>int*</td> <td>VISLongLong*</td> </tr> <tr> <td>unsigned int*</td> <td>VISULongLong*</td> </tr> <tr> <td>long*</td> <td>wchar_t*</td> </tr> </table>	char*	unsigned long*	unsigned char*	float*	short*	double*	unsigned short*	long double*	int*	VISLongLong*	unsigned int*	VISULongLong*	long*	wchar_t*
char*	unsigned long*														
unsigned char*	float*														
short*	double*														
unsigned short*	long double*														
int*	VISLongLong*														
unsigned int*	VISULongLong*														
long*	wchar_t*														
size	The number of the specified data types to be retrieved.														

```
virtual VISistream& getCString(char* data, unsigned maxlen);
```

This method allows you to retrieve a character string from the buffer at the current location. It returns a pointer to the location within the buffer immediately following the end of the data that was just retrieved.

Parameter	Description
data	The location where the retrieved character string is to be stored.
maxlen	The maximum number of characters to be retrieved.

```
virtual int is_available(unsigned long size);
```

Returns 1 if the specified size is less than or equal to the size of the buffer associated with this object.

Parameter	Description
size	Number of bytes that need to fit within this buffer.

```
virtual CORBA::ULong length() const;
```

Returns the total number of bytes in this object's buffer.

```
virtual void new_encapsulation() const;
```

Resets the starting offset within the buffer to 0.

```
void release_flag(CORBA::Boolean val);
```

Enables or disables the automatic freeing of buffer memory when this object is destroyed.

Parameter	Description
val	If val is set to <code>TRUE</code> , the buffer memory for this object will be freed when this object is destroyed. If val is set to <code>FALSE</code> , the buffer will not be freed when this object is destroyed.

```
CORBA::Boolean release_flag() const;
```

Returns `TRUE` if the automatic freeing of this object's buffer memory is enabled, otherwise `FALSE` is returned.

```
void reset();
```

Resets the starting offset, current offset and seek position to zero.

```
void rewind();
```

Resets the seek position to zero.

```
CORBA::ULong seekpos(CORBA::ULong pos);
```

Sets the current offset to the value contained in `pos`. If `pos` specifies an offset that is greater than the size of the buffer, a `CORBA::BAD_PARAM` exception is raised.

```
static CORBA::MarshalInBuffer *_duplicate(CORBA::MarshalInBuffer_ptr ptr);
```

Returns a duplicate pointer to the object pointed to by `ptr` and increments this object's reference count.

```
static CORBA::MarshalInBuffer *_nil();
```

Returns a `NULL` pointer of type `CORBA::MarshalInBuffer`.

```
static void _release(CORBA::MarshalInBuffer_ptr ptr);
```

Reduces the reference count of the object pointed to by `ptr`. When the reference count reaches 0, the object is destroyed. If the object's `release_flag` was set to `TRUE` when the object was constructed, the buffer associated with the object is freed.

CORBA::MarshalInBuffer operators

```
virtual VISostream&operator>>(<data_type> data);
```

This stream operator allows you retrieve a sequence of data of the specified source `data_type` at the current location.

This method returns a pointer to the location within the buffer immediately following the end of the data that was just written.

Parameter	Description
<code>data</code>	The data to be written to the buffer. The supported source data types are:
	<code>char*&</code> <code>long&</code>
	<code>char&</code> <code>unsigned long&</code>
	<code>unsigned char&</code> <code>float&</code>
	<code>short&</code> <code>double&</code>
	<code>unsigned short&</code> <code>long double&</code>
	<code>int&</code> <code>wchar_t*&</code>
	<code>unsigned int&</code> <code>wchar_t&</code>

CORBA::MarshalOutBuffer

```
class CORBA::MarshalOutBuffer : public VISostream
```

This class represents a stream buffer that allows IDL types to be written to a buffer and may be used by interceptor methods that you implement. See [“Portable Interceptor interfaces and classes” on page 213](#) for more information on the interceptor interfaces.

The `CORBA::MarshalOutBuffer` class is used on the client side to marshal the data associated with an operation request. It is used on the server side to marshal the data associated with a reply message. This class provides a wide range of methods for adding various types of data to the buffer or for retrieving what was written from the buffer.

This class provides several static methods for testing and manipulating `CORBA::MarshalOutBuffer` pointers.

A `CORBA::MarshalOutBuffer_var` class is also offered. It provides a wrapper that automatically manages the contained object.

Include file

The `mbuf.h` file should be included when you use this class. This file gets included in `corba.h`. So, you don't have to separately include `mbuf.h`.

CORBA::MarshalOutBuffer constructors/destructors

```
CORBA::MarshalOutBuffer(CORBA::ULong initial_size = 255, CORBA::Boolean
release_flag = 0);
```

Creates a `marshalOutBuffer` of size `initial_size`. The `MarshalOutBuffers` are capable of resizing themselves during a put operation. When there is not enough space in the buffer to hold all the data written to it, the size of the buffer doubles.

Parameter	Description
<code>initial_size</code>	The initial size of the buffer associated with this object. The default size is 255 bytes.
<code>release_flag</code>	If set to <code>TRUE</code> , the memory associated with <code>read_buffer</code> is freed when this object is destroyed. The default value is <code>FALSE</code> .

```
CORBA::MarshalOutBuffer(char *read_buffer, CORBA::ULong len, CORBA::Boolean
release_flag=0);
```

Creates an object with the specified buffer, buffer length and release flag value.

Parameter	Description
<code>read_buffer</code>	The buffer where the marshalled data will actually be stored.
<code>length</code>	The maximum number of bytes that may be stored in <code>read_buffer</code> .
<code>release_flag</code>	If set to <code>TRUE</code> , the memory associated with <code>read_buffer</code> is freed when this object is destroyed. The default value is <code>FALSE</code> .

```
virtual ~CORBA::MarshalOutBuffer();
```

This is the default destructor. The buffer memory associated with this object is released if the `release_flag` is set to `TRUE`. The `release_flag` may be set when the object is created or by invoking the `release_flag` method, described in [“CORBA::Boolean release_flag\(\) const;” on page 274](#).

CORBA::MarshalOutBuffer methods

```
char *buffer() const;
```

Returns a pointer to the buffer associated with this object.

```
CORBA::ULong curoff() const;
```

Returns the current offset within the buffer associated with this object.

```
virtual CORBA::ULong length() const;
```

Returns the total number of bytes in this object's buffer.

```
virtual void new_encapsulation() const;
```

Resets the starting offset within the buffer to zero.

```
virtual VISostream& put(char data);
```

Adds a single character to the buffer at the current location.

This method returns a pointer to the location within the buffer immediately following the end of the data that was just added.

Parameter	Description
data	The char to be stored.

```
virtual VISostream& put(const <data_type> data, unsigned size);
```

These methods allow you to store a sequence of data in the buffer at the current location.

This method returns a pointer to the location within the buffer immediately following the end of the data that was just added.

Parameter	Description														
data	The data is to be stored. The supported source data types are: <table border="0"> <tr> <td>char*</td> <td>unsigned long*</td> </tr> <tr> <td>unsigned char*</td> <td>float*</td> </tr> <tr> <td>short*</td> <td>double*</td> </tr> <tr> <td>unsigned short*</td> <td>long double*</td> </tr> <tr> <td>int*</td> <td>VISLongLong*</td> </tr> <tr> <td>unsigned int*</td> <td>VISULongLong*</td> </tr> <tr> <td>long*</td> <td>wchar_t*</td> </tr> </table>	char*	unsigned long*	unsigned char*	float*	short*	double*	unsigned short*	long double*	int*	VISLongLong*	unsigned int*	VISULongLong*	long*	wchar_t*
char*	unsigned long*														
unsigned char*	float*														
short*	double*														
unsigned short*	long double*														
int*	VISLongLong*														
unsigned int*	VISULongLong*														
long*	wchar_t*														
size	The number of the specified data types to be stored.														

```
virtual VISostream& putCString(const char* data);
```

This method allows you to store a character string into the buffer at the current location. It returns a pointer to the location within the buffer immediately following the end of the data that was just added.

Parameter	Description
data	The character string to be stored.

```
void release_flag(CORBA::Boolean val);
```

Enables or disables the automatic freeing of buffer memory when this object is destroyed.

Parameter	Description
val	If val is set to TRUE, the buffer memory for this object will be freed when this object is destroyed. If val is set to FALSE, the buffer will not be freed when this object is destroyed

```
CORBA::Boolean release_flag() const;
```

Returns TRUE if the automatic freeing of this object's buffer memory is enabled, otherwise returns FALSE.

```
void reset();
```

Resets the starting offset, current offset and seek position to zero.

```
void rewind();
```

Resets the seek position to zero.

```
CORBA::ULong seekpos(CORBA::ULong pos);
```

Sets the current offset to the value contained in `pos`. If `pos` specifies an offset that is greater than the size of the buffer, a `CORBA::BAD_PARAM` exception is raised.

```
static CORBA::MarshalOutBuffer *_duplicate(CORBA::MarshalOutBuffer_ptr ptr);
```

Returns a duplicate pointer to this object pointed to by `ptr` and increments this object's reference count.

```
static CORBA::MarshalOutBuffer *_nil();
```

Returns a `NULL` pointer of type `CORBA::MarshalOutBuffer`.

```
static void _release(CORBA::MarshalOutBuffer_ptr ptr);
```

Reduces the reference count of the object pointed to by `ptr`. If the reference count is then zero, the object is destroyed. If the object's `release_flag` was set to `TRUE` when it was constructed, the buffer associated with the object is freed.

CORBA::MarshalOutBuffer operators

```
virtual VISostream& operator<<(<data_type> data);
```

This stream operator allows you to add data of the specified `data_type` to the buffer at the current location.

This method returns a pointer to the location within the buffer immediately following the end of the data that was just written.

Parameter	Description
<code>data</code>	The data to be obtained to the buffer. The supported data types are:
<code>const char*</code>	<code>unsigned long</code>
<code>char</code>	<code>float</code>
<code>unsigned char</code>	<code>double</code>
<code>short</code>	<code>long double</code>
<code>unsigned short</code>	<code>VISLongLong</code>
<code>int</code>	<code>VISULongLong</code>
<code>unsigned int</code>	<code>wchar_t*</code>
<code>long</code>	<code>wchar_t</code>

Chapter 17

Location service interfaces and classes

This section describes the interfaces you can use to locate object instances on a network of Smart Agents. For more information on the Location Service, go to the *VisiBroker for C++ Developer's Guide*, Using the Location Service section.

Agent

```
class Agent : public CORBA::Object
```

This class provides methods that enable you to locate all instances of a particular object on a network of Smart Agents. The methods offered by this class are divided into two categories: those that query a Smart Agent for data about objects and those that deal with *triggers*.

Your client application can obtain object information based on an interface repository ID alone or in combination with an instance name.

Triggers allow your client application to be notified of changes in the availability of one or more object instances.

IDL definition

```

interface Agent {
    HostnameSeq all_agent_locations()
        raises (Fail);
    RepositoryIdSeq all_repository_ids()
        raises (Fail);
    ObjSeqSeq all_available()
        raises (Fail);
    ObjSeq all_instances (in string repository_id)
        raises (Fail);
    ObjSeq all_replica (in string repository_id, in string instance_name)
        raises (Fail);
    DescSeqSeq all_available_descs()
        raises (Fail);
    DescSeq all_instances_descs (in string repository_id)
        raises (Fail);
    DescSeq all_replica_descs (in string repository_id, in string instance_name)
        raises (Fail);
    void reg_trigger(in TriggerDesc desc, in TriggerHandler handler)
        raises (Fail);
    void unreg_trigger(in TriggerDesc desc, in TriggerHandler handler)
        raises (Fail);
    attribute boolean willRefreshOADs;
};

```

Include file

You should include the **locate_c.hh** file when you use this class.

Agent methods

`ObjLocation::HostnameSeq_ptr all_agent_locations();`

Returns a sequence of host names representing the hosts on which osagent processes are currently executing.

This method throws the following exceptions:

Exception	Description
Fail	The <code>FailReason</code> values that may be presented include: <code>NO_AGENT_AVAILABLE</code> , <code>NO_SUCH_TRIGGER</code> , <code>AGENT_ERROR</code> . For more information on the <code>Fail</code> class, see “Fail” on page 285 .

See also:

- [“<type>Seq” on page 287](#)


```
ObjLocation::ObjSeqSeq all_available();
```

Returns a sequence of object references for all objects currently registered with some Smart Agent on the network.

This method throws the following exceptions:

Exception	Description
Fail	The <code>FailReason</code> values that may be presented include: <code>NO_AGENT_AVAILABLE</code> , <code>NO_SUCH_TRIGGER</code> , <code>AGENT_ERROR</code> . For more information on the <code>Fail</code> class, see “Fail” on page 285 .

See also:

- [“<type>Seq” on page 287](#)

```
ObjLocation::DescSeqSeq_ptr all_available_descs();
```

Returns descriptions for all objects currently registered with a Smart Agent on the network. The description information returned is organized by repository id.

This method throws the following exceptions:

Exception	Description
Fail	The <code>FailReason</code> values that may be presented include: <code>NO_AGENT_AVAILABLE</code> , <code>NO_SUCH_TRIGGER</code> , <code>AGENT_ERROR</code> . For more information on the <code>Fail</code> class, see “Fail” on page 285 .

See also:

- [“<type>Seq” on page 287](#)

```
ObjLocation::ObjSeq_ptr all_instances(const char *repository_id);
```

Returns a sequence of object references to all instances with the specified `repository_id`.

Parameter	Description
<code>repository_id</code>	The repository ID of the object references to be retrieved.

This method throws the following exceptions:

Exception	Description
Fail	Any of the <code>FailReason</code> values, other than <code>NO_SUCH_TRIGGER</code> , may be presented. For more information on the <code>Fail</code> class, see “Fail” on page 285 .

See also:

- [“<type>Seq” on page 287](#)

```
ObjLocation::DescSeq_ptr all_instances_descs(const char *repository_id);
```

Returns description information for all object instances with the specified repository_id.

Parameter	Description
repository_id	The repository ID of the object descriptions to be retrieved.

This method throws the following exceptions:

Exception	Description
Fail	Any of the <code>FailReason</code> values, other than <code>NO_SUCH_TRIGGER</code> , may be presented. For more information on the <code>Fail</code> class, see “Fail” on page 285 .

See also:

- [“<type>Seq” on page 287](#)

```
ObjLocation::ObjSeq_ptr all_replica(const char *repository_id, const char *instance_name);
```

Returns a sequence of object references for objects with the specified repository_id and instance_name.

Parameter	Description
repository_id	The repository ID of the object references to be retrieved.
instance_name	The instance name of the object references to be returned.

This method throws the following exceptions:

Exception	Description
Fail	Any of the <code>FailReason</code> values, other than <code>NO_SUCH_TRIGGER</code> , may be presented. For more information on the <code>Fail</code> class, see “Fail” on page 285 .

See also:

- [“<type>Seq” on page 287](#)

```
ObjLocation::DescSeq_ptr all_replica_descs(const char *repository_id, const char *instance_name);
```

Returns a sequence of description information for all object instances with the specified repository_id and instance_name.

Parameter	Description
repository_id	The repository ID of the object descriptions to be retrieved.
instance_name	The instance name of the object descriptions to be retrieved.

This method throws the following exceptions:

Exception	Description
Fail	Any of the <code>FailReason</code> values, other than <code>NO_SUCH_TRIGGER</code> , may be presented. For more information on the <code>Fail</code> class, see “Fail” on page 285 .

See also

- [“<type>Seq” on page 287](#)

```
CORBA::StringSequence* all_repository_ids();
```

This method retrieves all interfaces known to any osagent. This method throws the following exception:

Exception	Description
Fail	The repository id is invalid.

```
void reg_trigger(const ObjLocation::TriggerDesc& desc,
ObjLocation::TriggerHandler_ptr hdlr);
```

Registers the trigger handler `hdlr` for object instances that match the description information specified in `desc`.

Note

A `TriggerHandler` is invoked every time an object that satisfies the trigger's description becomes available. If you are only interested in learning when the first instance of the object becomes available, you should use the `unreg_trigger` method to remove the trigger after the first notification is received.

Parameter	Description
<code>desc</code>	The object instance description information, which can contain combinations of the following information: repository ID, instance name, hostname. You can provide more or less information to narrow or widen the object instances to be monitored.
<code>hdlr</code>	The trigger handler object being registered.

This method throws the following exceptions:

Exception	Description
Fail	Any of the <code>FailReason</code> values, other than <code>NO_SUCH_TRIGGER</code> , may be presented. For more information on the <code>Fail</code> class, see “Fail” on page 285 .

```
void unreg_trigger(const ObjLocation::TriggerDesc& desc,
ObjLocation::TriggerHandler_ptr hdlr);
```

Unregisters the trigger handler `hdlr` for object instances that match the description information specified in `desc`.

Parameter	Description
<code>desc</code>	The object description information.
<code>hdlr</code>	The trigger handler object being unregistered.

This method throws the following exceptions:

Exception	Description
Fail	The only <code>FailReason</code> value possible is <code>NO_SUCH_TRIGGER</code> . For more information on the <code>Fail</code> class, see “Fail” on page 285 .

```
CORBA::Boolean willRefreshOADs();
```

Returns `TRUE` if the set of Object Activation Daemon is updated each time a method offered by this class is invoked, otherwise returns `FALSE`. If the cache is not refreshed on each invocation, the following conditions may occur:

- All objects are still reported, but their descriptor's `activable` flag may be incorrect.
- Any attempt to verify the existence of an object registered with an OAD that has been started since the last refresh of the OAD cache causes those objects to be activated by the OAD.

Desc

```
void willRefreshOADs(CORBA::Boolean val);
```

This class maintains a set of Object Activation Daemons. This method enables or disables the automatic refreshing of the OADs contained in this set.

Parameter	Description
val	If <code>TRUE</code> , the OAD set will be refreshed whenever a method offered by this class is invoked.

Desc

```
struct Desc
```

This structure contains information you use to describe the characteristics of an object. You pass this structure as an argument to several of the Location Service methods described in the chapter. The *Desc* structure, or a sequence of them, is returned by some of the Location Service methods.

See also

- [“<type>Seq” on page 287](#)

IDL definition

```
module ObjLocation {
    struct Desc {
        Object ref;
        IIOP::ProfileBody iiop_locator;
        string repository_id;
        string instance_name;
        boolean activable;
        string agent_hostname;
    };
    :
};
```

Desc members

Object **ref**

A reference to the object being described.

IIOP::ProfileBody **iiop_locator**

Represents profile data for the object, described in [“IIOP::ProfileBody” on page 269](#).

CORBA::String_var **repository_id**

The object's repository identifier.

CORBA::String_var **instance_name**

The object's instance name.

CORBA::Boolean **activable**

Set to `TRUE` to indicate that this object is registered with the Object Activation Daemon. It is set to `FALSE` to indicate that the object was started manually and is registered with the `osagent`.

CORBA::String_var **agent_hostname**

The name of the host running the Smart Agent with which this object is registered.

Fail

```
class Fail : public CORBA::UserException
```

This exception class may be thrown by the `Agent` class to indicate various errors. The data member `FailReason` is used to indicate the nature of the failure.

Fail members

FailReason **reason**

Set to one of the following values to indicate the nature of the failure:

```
enum FailReason {
    NO_AGENT_AVAILABLE,
    INVALID_REPOSITORY_ID,
    INVALID_OBJECT_NAME,
    NO_SUCH_TRIGGER,
    AGENT_ERROR
};
```

TriggerDesc

```
struct TriggerDesc
```

This structure contains information you use to describe the characteristics of one or more objects for which you wish to register a `TriggerHandler`, described in [“TriggerHandler” on page 286](#).

The `host_name` and `instance_name` members may be set to `NULL` to monitor the widest possible set of objects. The more information specified, the smaller the set of objects is.

IDL definition

```
module ObjLocation {
    :
    struct TriggerDesc {
        string repository_id;
        string instance_name;
        string host_name;
    };
    :
};
```

TriggerDesc members

CORBA::String_var **repository_id**

The repository identifiers of the objects to be monitored by the `TriggerHandler`.

CORBA::String_var **instance_name**

The instance name of the object to be monitored by the `TriggerHandler`. May be set to `NULL` to include all possible instance names.

CORBA::String_var **host_name;**

The host name where the object or objects monitored by the `TriggerHandler` are located. May be set to `NULL` to include all hosts in the network.

TriggerHandler

You use this base class to derive your own callback object to be invoked every time an object becomes available or unavailable. You specify the criteria for the object or objects in which you are interested. You register your `TriggerHandler` object using the `Agent::reg_trigger` method, described in the “void `reg_trigger`(const `ObjLocation::TriggerDesc& desc`, `ObjLocation::TriggerHandler_ptr hdlr`);” on page 283.

You must provide implementations for the `impl_is_ready` and `impl_is_down` methods.

IDL definition

```
interface TriggerHandler {
    void impl_is_ready(in Desc desc);
    void impl_is_down(in Desc desc);
};
```

Include file

You should include the `locate_c.hh` file when you use this class.

TriggerHandler methods

```
virtual void impl_is_ready(const Desc& desc);
```

This method is invoked by the Location Service when an object instance matching the criteria specified in `desc` becomes accessible.

Parameter	Description
<code>desc</code>	The object description information.

```
virtual void impl_is_down(const Desc& desc);
```

This method is invoked by the Location Service when an object instance matching the criteria specified in `desc` is no longer accessible.

Parameter	Description
<code>desc</code>	The object description information.

<type>Seq

This is a generalized class description for the following sequence classes used by the Location Service:

Class	Description
DescSeq	A sequence of Desc structures.
HostnameSeq	A sequence of host names.
ObjSeq	A sequence of object references.
RepositoryIdSeq	A sequence of repository identifiers.

Each class represents a particular sequence of <type>. The Location Service returns lists of information to your client application in the form of sequences which are mapped to one of these classes.

Each class offers operators for indexing items in the sequence just as you would a C++ array. Each class also offers methods for setting and obtaining the length of the array.

The code sample below shows the correct way to index a HostnameSeq returned from the Agent::all_agent_locations method.

```

:
ObjLocation::HostnameSeq_var hostnames(myAgent->all_agent_locations());
for (CORBA::ULong i=0; i < hostnames->length(); i++) {
    cout << "Agent host #" << i+1 << ": " << hostnames[i] << endl;
}
:

```

See also

- “<type>SeqSeq” on page 288

<type>Seq methods

<type>& operator[](CORBA::ULong index) const;

Returns a reference to the element in the sequence identified by index.

Caution

You must use a CORBA::ULong type for the index. Using an int type may lead to unpredictable results.

Parameter	Description
index	The zero-based index of the element to be returned.

This method throws the following exception:

Exception	Description
CORBA::BAD_PARAM	The index specified is less than zero or greater than the size of the sequence.

CORBA::ULong length() const;

Returns the number of elements in the sequence.

<type>SeqSeq

```
void length(CORBA::ULong len);
```

Sets the maximum length of the sequence to the value contained in `len`.

Parameter	Description
<code>len</code>	The new length for the sequence.

<type>SeqSeq

This is a generalized class description for the following classes used by the Location Service:

Class	Description
<code>DescSeqSeq</code>	A sequence of <code>DescSeq</code> objects.
<code>ObjSeqSeq</code>	A sequence of <code>ObjSeq</code> objects.

Each class represents a particular sequence of <type>Seq. Some Location Service methods return lists of information to your client application in the form of sequences of sequences which are mapped to one of these classes.

Each class offers operators for indexing items in the sequence just as you would a C++ array. The class also offer methods for setting and obtaining the length of the array.

See also

- [“<type>Seq” on page 287](#)

<type>SeqSeq methods

```
<type>Seq& operator[] (CORBA::ULong index) const;
```

Returns a reference to the element in the sequence identified by `index`. The reference is to a one dimensional sequence, described in [“<type>Seq” on page 287](#).

Caution

You must use a `CORBA::ULong` type for the index. Using an `int` type may lead to unpredictable results.

Parameter	Description
<code>index</code>	The zero-based index of the element to be returned.

This method throws the following exceptions:

Exception	Description
<code>CORBA::BAD_PARAM</code>	The index specified is less than zero or greater than the size of the sequence.

```
CORBA::ULong length() const;
```

Returns the number of elements in the sequence.

```
void length(CORBA::ULong len);
```

Sets the maximum length of the sequence to the value contained in `len`.

Parameter	Description
<code>len</code>	The new length for the sequence.

Initialization interfaces and classes

This section describes the interfaces and classes that are provided for statically initializing VisiBroker ORB services such as interceptors.

VISInit

class `VISInit`

This abstract base class provides for the static initialization of service classes after the VisiBroker ORB and BOA have been initialized. By deriving your service class from `VISInit` and declaring it statically, you ensure that your service class instance will be properly initialized.

The VisiBroker ORB invokes the `VISInit::ORB_init` and `VISInit::BOA_init` whenever the application calls `CORBA::ORB_init` or `BOA_init` methods. By providing your own implementations of these methods, you may add any needed initialization that must be performed for your service.

Include file

Include the `vinit.h` file when you use this class.

VISInit constructors/destructors

`VISInit()`;

This is the default constructor.

`VISInit(CORBA::Long init_priority)`;

This constructor creates a `VISInit`-derived object with the specified priority, which determines when it will be initialized relative to other `VISInit`-derived objects.

Internal VisiBroker classes which need to be initialized before user-defined classes have a negative priority value. The lowest priority value currently used by VisiBroker internal classes is `-10`.

Note You should set a priority value less than -10 if your class must be initialized before the VisiBroker internal classes.

If no priority value is specified, the default value is 0, which means that the class will be initialized after the internal VisiBroker classes.

Parameter	Description
<code>init_priority</code>	The initialization priority for this object. A negative priority value causes this class to be initialized earlier. A positive priority value causes this class to be initialized later.

```
virtual ~VISInit();
```

This is the default destructor.

VISInit methods

```
virtual void ORB_init(int& argc, char * const *argv, CORBA::ORB_ptr orb);
```

This method will be called during VisiBroker ORB initialization. Your implementation should provide for the initialization of the client-side interceptor factory that you wish to use.

Parameter	Description
<code>argc</code>	The count of arguments.
<code>argv</code>	An array of argument pointers.
<code>orb</code>	The VisiBroker ORB being initialized.

```
virtual void ORB_initialized(CORBA::ORB_ptr orb);
```

This method will be called after the VisiBroker ORB is initialized. Your implementation should provide for the initialization of the client-side interceptor factory that you wish to use.

Parameter	Description
<code>orb</code>	The VisiBroker ORB being initialized.

```
virtual void BOA_init(int& argc, char * const *argv, CORBA::BOA_ptr boa);
```

This method will be called when the BOA is initialized. Your implementation should provide for the initialization of the server-side interceptor factory that you wish to use.

Parameter	Description
<code>argc</code>	The count of arguments.
<code>argv</code>	An array of argument pointers.
<code>boa</code>	The BOA being initialized.

```
virtual void ORB_shutdown();
```

This method will be called when the VisiBroker ORB is shut down.

Chapter 19

Real-Time CORBA interfaces and classes

This section describes the Real-Time CORBA interfaces and classes supported by VisiBroker for C++.

Note Before using these interfaces, read ““Real-Time CORBA Extensions” in the *VisiBroker for C++ Developer's Guide* for descriptions and usage information on the supported extensions.

Introduction

Real-Time CORBA provides a set of APIs that support the development of predictable CORBA-based systems, through the control of the number and priority of threads involved in the execution of CORBA invocations.

The majority of the Real-Time CORBA API is specified in IDL, and is mapped to C++ according to the rules of the CORBA C++ language mapping. The Real-Time CORBA IDL is scoped within module RTCORBA, and hence the C++ class names are all prefixed `RTCORBA::`.

The following Real-Time CORBA interfaces and classes are described in the sections that follow :

- `RTCORBA::Current`
- `RTCORBA::Mutex`
- `RTCORBA::NativePriority`
- `RTCORBA::Priority`
- `RTCORBA::PriorityMapping`
- `RTCORBA::PriorityModel`
- `RTCORBA::PriorityModelPolicy`
- `RTCORBA::RTORB`
- `RTCORBA::ThreadPoolId`
- `RTCORBA::ThreadPoolPolicy`

Include file

To use any of the Real-Time CORBA features described in this chapter, the application should include the file `rtcorba.h`, which is one of the include files supplied with VisiBroker for C++.

RTCORBA::Current

```
class RTCORBA::Current : public CORBA::Object
typedef RTCORBA::Current* Current_ptr
class RTCORBA::Current_var
```

The class `RTCORBA::Current` provides methods that allow a Real-Time CORBA Priority value to be associated with the current thread of execution, and the reading of the Real-Time CORBA Priority value presently associated with the current thread.

When a Real-Time CORBA Priority value is associated with the current thread, that value is immediately used to set the Native Priority of the underlying thread. The Native Priority value to apply to the thread is obtained by means of the currently installed Priority Mapping.

Where the Client Propagated Priority Model is in use, the Priority associated with a thread also determines the priority of CORBA invocations made from that thread. For details, see “Real-Time CORBA Priority Models” in the *VisiBroker for C++ Developer's Guide*.

`RTCORBA::Current` is defined in IDL, as a locality constrained interface. Hence applications handle `RTCORBA::Current` by means of CORBA Object References, using the C++ classes `RTCORBA::Current_ptr` and `RTCORBA::Current_var`.

See “[RTCORBA::Priority](#)” on page 295 for more information.

RTCORBA::Current Creation and Destruction

`RTCORBA::Current` is a special interface. Applications need not be concerned with which instance of it they are dealing. A reference to `RTCORBA::Current` is obtained through the `resolve_initial_references` method of `RTCORBA::RTORB`, and is released in the normal way when it is no longer required. For details see “Real-Time CORBA Current” in the *VisiBroker for C++ Developer's Guide*.

IDL definition

```
//Locality Constrained Object
interface Current {
    attribute Priority base_priority;
};
```

RTCORBA::Current methods

```
void base_priority(Priority _val);
```

Associates the `RTCORBA::Priority` value `_val` with the current thread of execution.

Parameter	Description
<code>_val</code>	The Priority value to associate with the thread

```
Priority base_priority();
```

Gets the `RTCORBA::Priority` value associated with the current thread of execution.

RTCORBA::Mutex

```
class RTCORBA::Mutex : public CORBA::Object
typedef RTCORBA::Mutex* RTCORBA::Mutex_ptr
class RTCORBA::Mutex_var
class TimeBase {
    typedef unsigned long long TimeT;
};
```

The interface `RTCORBA::Mutex` provides applications with a mutex synchronization primitive that is guaranteed to have the same priority inheritance properties as mutexes used internally by VisiBroker to protect ORB resources. For details, see “Real-Time CORBA Mutex API” in the *VisiBroker for C++ Developer's Guide*.

`RTCORBA::Mutex` is defined in IDL, as a locality constrained interface. Hence applications handle `RTCORBA::Mutex` instances by means of CORBA Object References, using the C++ classes `RTCORBA::Mutex_ptr` and `RTCORBA::Mutex_var`.

See “[RTCORBA::RTORB](#)” on [page 298](#) for more information.

Mutex Creation and Destruction

A new `RTCORBA::Mutex` is obtained using the **create_mutex** operation of the `RTCORBA::RTORB` interface. The new `RTCORBA::Mutex` is created in an unlocked state.

When the `RTCORBA::Mutex` is no longer needed, it is destroyed using the **destroy_mutex** operation of `RTCORBA::RTORB`. See “[RTCORBA::RTORB](#)” on [page 298](#) for details.

Note that if the `RTCORBA::Mutex_var` type is used in place of the `RTCORBA::Mutex_ptr` type, the reference is automatically released when the `_var` instance goes out of scope, but the `RTCORBA::Mutex` instance it refers to is not automatically destroyed. The `RTCORBA::Mutex` instance must still be destroyed with a call to **destroy_mutex**.

IDL definition

```
// Locality Constrained Object
interface Mutex {
    void lock();
    void unlock();
    boolean try_lock ( in TimeBase::TimeT max_wait );
};

interface RTORB {
    :
    Mutex create_mutex();
    void destroy_mutex( in Mutex the_mutex );
    :
};

// defined in TimeBase.idl
module TimeBase {
    typedef unsigned long long TimeT;
};
```

RTCORBA::Mutex Methods

```
void lock();
```

Locks the `RTCORBA::Mutex`. When the `RTCORBA::Mutex` object is in the unlocked state, the first thread to call the `lock()` operation causes the `Mutex` object to change to the locked state. Subsequent threads that call the `lock()` operation while the `Mutex` object is still in the locked state will block until the owner thread unlocks it.

```
void unlock();
```

Unlocks the locked `RTCORBA::Mutex`.

```
CORBA::Boolean try_lock( const TimeBase::TimeT _max_wait );
```

Attempts to lock the `RTCORBA::Mutex`, waiting for a maximum of `_max_wait` amount of time. Returns true if the lock is successfully taken within the time, or false if it could not be taken before the time expired.

Parameter	Description
<code>_max_wait</code>	The maximum amount of time to wait for the lock, in 100 nanosecond ticks. A value of 0 means do not wait for the lock.

RTCORBA::NativePriority

```
typedef CORBA::Short RTCORBA::NativePriority
```

The type `RTCORBA::NativePriority` is used to represent priorities in the priority scheme of the particular Operating System that the Real-Time ORB is running on. Real-Time CORBA applications only use `RTCORBA::NativePriority` values in special circumstances:

- When defining a Priority Mapping. For details, see [“RTCORBA::PriorityMapping” on page 295](#).
- When interacting directly with the Operating System, or with some other non-CORBA subsystem, that works in terms of Native Priorities. This should still be done by means of the installed Priority Mapping. For details, see “Using Native Priorities in VisiBroker Application Code” in the *VisiBroker for C++ Developer's Guide*.

Normally, within a Real-Time CORBA application, priorities are expressed in terms of `RTCORBA::Priority` values.

IDL definition

```
typedef CORBA::Short NativePriority;
```

RTCORBA::Priority

```
typedef CORBA::Short RTCORBA::Priority
static const Priority RTCORBA::minPriority; // 0
static const Priority RTCORBA::maxPriority; // 32767
```

The type `RTCORBA::Priority` should be used to represent priority values in a Real-Time CORBA application. These values are mapped on to the Native Priority scheme of the particular Operating System that the application is running on by the currently installed Priority Mapping. For a detailed discussion of Real-Time CORBA Priority, see “Real-Time CORBA Priority” in the *VisiBroker for C++ Developer's Guide*.

The only time a Real-Time CORBA application should use Native Priority values is when interacting directly with the Operating System or some other non-CORBA subsystem. Even then, this should still be done using the installed Priority Mapping. For details see 'Using Native Priorities in VisiBroker Application Code' in the *VisiBroker for C++ Developer's Guide*.

`RTCORBA::Priority` values are in the range 0 to 32767. However, it is not expected that this full range of priorities will be used in a Real-Time CORBA system. Instead, the application system designer should decide on a suitable range of priorities for that system, and implement a Priority Mapping that only allows priority values in that range. For many applications the default valid range of 0 to 31 is acceptable, but there might still be reasons to override the default Priority Mapping. See “[RTCORBA::PriorityMapping](#)” on page 295 for details.

IDL definition

```
typedef CORBA::Short Priority;
static const Priority minPriority; // 0
static const Priority maxPriority; // 32767
```

RTCORBA::PriorityMapping

```
class RTCORBA::PriorityMapping
```

The `RTCORBA::PriorityMapping` class facilitates the mapping of `RTCORBA::Priority` values to and from the Native Priority scheme of the Operating System the Real-Time ORB is running on. The ORB calls out to a Priority Mapping object whenever it needs to map a `RTCORBA::Priority` value to a `RTCORBA::NativePriority` value or vice versa.

A Real-Time CORBA application should describe its priorities in terms of `RTCORBA::Priority` values. However, the application might need to make explicit use of the installed Priority Mapping, in order to interact directly with the Operating System or some other non-CORBA subsystem. For details see “Using Native Priorities in VisiBroker Application Code” in the *VisiBroker for C++ Developer's Guide*.

The range of `RTCORBA::Priority` values supported by a Priority Mapping should always start from zero. The Real-Time ORB expects `RTCORBA::Priority` zero to be valid. Also, this convention makes integration of different Real-Time CORBA systems on the same node easier.

PriorityMapping Creation and Destruction

It is not necessary to create instances of a Priority Mapping in the code of a normal Real-Time CORBA application. The available Priority Mapping is automatically used by the ORB, and can be accessed by the application if necessary.

Exactly one Priority Mapping is 'installed' at any one time. A 'default' Priority Mapping is provided, which is installed by default. This Default Priority Mapping can be overridden by installing an application-implemented Priority Mapping object. The installation process is described in the section “Replacing the Default Priority Mapping” in the *VisiBroker for C++ Developer's Guide*.

IDL definition

```
// 'native' IDL type
native PriorityMapping;
```

The `RTCORBA::PriorityMapping` IDL type is defined as a 'native' IDL type. This means that its mapping to different programming languages is defined on a per-language basis. The C++ class representing `RTCORBA::PriorityMapping` has the following declaration:

```
class PriorityMapping {
public:
    virtual CORBA::Boolean to_native(
        RTCORBA::Priority corba_priority,
        RTCORBA::NativePriority &native_priority )=0;
    virtual CORBA::Boolean to_CORBA(
        RTCORBA::NativePriority native_priority,
        RTCORBA::Priority &corba_priority )=0;
    virtual RTCORBA::Priority max_priority() = 0;
    PriorityMapping();
    virtual ~PriorityMapping() {}
    static RTCORBA::PriorityMapping * instance();
};
```

The purpose of each method is explained in the next section, “PriorityMapping Methods”.

PriorityMapping Methods

```
static RTCORBA::PriorityMapping * instance();
```

This static method, implemented by VisiBroker for C++, can be used by Real-Time CORBA applications to access the currently installed Priority Mapping. For details see “Using Native Priorities in VisiBroker Application Code” in the *VisiBroker for C++ Developer's Guide* for details.

```
virtual RTCORBA::Priority max_priority() = 0;
```

This method returns the maximum Real-Time CORBA Priority value that is valid using this Priority Mapping. For example, if the installed Priority Mapping maps Real-Time CORBA Priorities in the range 0 to 31, the value 31 will be returned every time this method is called.

This method must be implemented when implementing a new Priority Mapping.


```
virtual CORBA::Boolean to_CORBA (
    RTCORBA::NativePriority native_priority,
    RTCORBA::Priority &corba_priority ) = 0;
```

This method maps a given Native Priority value, `native_priority`, to a Real-Time CORBA Priority value. If the Native Priority value is in the range supported by this Priority Mapping, the resultant Real-Time CORBA Priority value is stored in `corba_priority`, and a true value is returned. Otherwise `corba_priority` is not changed, and a false is returned.

This method must be implemented when implementing a new Priority Mapping..

Parameter	Description
<code>native_priority</code>	The Native Priority value to be mapped to a Real-Time CORBA Priority.
<code>corba_priority</code>	The variable to assign the mapped Real-Time CORBA Priority value to.

```
virtual CORBA::Boolean to_native (
    RTCORBA::Priority corba_priority,
    RTCORBA::NativePriority &native_priority ) = 0;
```

This method maps a given Real-Time CORBA Priority value, `corba_priority`, to a Native Priority value. If the Real-Time CORBA Priority value is in the range supported by this Priority Mapping, the resultant Native Priority value is stored in `native_priority`, and a true value is returned. Otherwise `native_priority` is not changed, and a false value is returned.

This method must be implemented when implementing a new Priority Mapping..

Parameter	Description
<code>corba_priority</code>	The Real-Time CORBA Priority value to be mapped to a Native Priority.
<code>native_priority</code>	The variable to assign the mapped Native Priority value to.

RTCORBA::PriorityModel

```
enum RTCORBA::PriorityModel {
    CLIENT_PROPAGATED,
    SERVER_DECLARED
};
```

This enumeration specifies the two Real-Time CORBA Priority Models : *Client Propagated Priority Model* and *Server Declared Priority Model*. These are described in the section “Real-Time CORBA Priority Models” in the *VisiBroker for C++ Developer’s Guide*

These enumeration values are used as values for a parameter to the `create_priority_model_policy` method of `RTCORBA::RTOB`. See “[RTCORBA::PriorityModelPolicy](#)” on page 298 for details.

RTCORBA::PriorityModelPolicy

```
class RTCORBA::PriorityModelPolicy : CORBA::Policy
```

An instance of this Real-Time Policy type is created by calling the `create_priority_model_policy` method of `RTCORBA::RTORB`. The Policy instance can then be used to configure a Real-Time POA at the time of its creation, by passing it into the `create_POA` method, as a member of the Policy List parameter.

See “[RTCORBA::RTORB](#)” on page 298 and “[RTCORBA::PriorityModel](#)” on page 297 for more information.

IDL definition

```
interface PriorityModelPolicy : CORBA::Policy {
    readonly attribute PriorityModel priority_model;
    readonly attribute Priority server_priority;
};
```

RTCORBA::RTORB

```
class RTCORBA::RTORB : public CORBA::Object
typedef RTCORBA::RTORB* RTCORBA::RTORB_ptr
class RTCORBA::RTORB_var
```

The interface `RTCORBA::RTORB` provides methods for the management of Real-Time CORBA Threadpools and Mutexes, and to create instances of Real-Time CORBA Policies.

`RTCORBA::RTORB` is defined in IDL, as a locality constrained interface. Hence applications handle `RTCORBA::RTORB` by means of CORBA Object References, using the C++ classes `RTCORBA::RTORB_ptr` and `RTCORBA::RTORB_var`.

Note As stated in the *VisiBroker for C++ Developer's Guide*, to support Real-Time CORBA Extensions, the VisiBroker for C++ ORB has to operate in a special 'real-time compatible' mode, the behavior and semantics of which differ from the regular mode of operation. Since obtaining an “RTORB” reference automatically puts the ORB in this special mode, you should **obtain an “RTORB” reference as early as possible** in your application code to avoid any possible inconsistency in behavior.

See “[RTCORBA::Mutex](#)” on page 293, “[RTCORBA::Priority](#)” on page 295, “[RTCORBA::ThreadpoolId](#)” on page 301, and “[RTCORBA::ThreadpoolPolicy](#)” on page 302. For details on the use of Real-Time CORBA Threadpools, see “Threadpools” in the *VisiBroker for C++ Developer's Guide*.

RTORB Creation and Destruction

The Real-Time ORB does not need to be explicitly initialized—it is initialized implicitly as part of the regular `CORBA::ORB_init` call.

To use the Real-Time ORB operations, the application must have a reference to the Real-Time ORB instance. This reference can be obtained any time after the call to `ORB_init`, and is obtained through the `resolve_initial_references` operation on `CORBA::ORB`, with the object ID string “RTORB” as the parameter. For details, see “Real-Time CORBA ORB” in the *VisiBroker for C++ Developer's Guide*.

IDL definition

```
// locality constrained interface
interface RTORB {
    Mutex create_mutex();
    void destroy_mutex( in Mutex the_mutex );

    exception InvalidThreadpool {};

    ThreadpoolId create_threadpool (
        in unsigned long stacksize,
        in unsigned long static_threads,
        in unsigned long dynamic_threads,
        in Priority default_priority,
        in boolean allow_request_buffering,
        in unsigned long max_buffered_requests,
        in unsigned long max_request_buffer_size );

    void destroy_threadpool( in ThreadpoolId threadpool )
        raises (InvalidThreadpool);

    void threadpool_idle_time( in ThreadpoolId threadpool,
        in unsigned long seconds )
        raises (InvalidThreadpool);

    PriorityModelPolicy create_priority_model_policy(
        in PriorityModel priority_model,
        in Priority server_priority );

    ThreadpoolPolicy create_threadpool_policy(
        in ThreadpoolId threadpool );
};
```

RTORB Methods

Mutex_ptr **create_mutex**();

Creates a new Real-Time CORBA Mutex and returns a reference to it.

void **destroy_mutex**(Mutex_ptr _the_mutex);

Destroys a Real-Time CORBA Mutex.

Parameter	Description
<u>_the_mutex</u>	Reference of the Mutex to destroy.

```
ThreadpoolId create_threadpool(
    CORBA::ULong _stacksize,
    CORBA::ULong _static_threads,
    CORBA::ULong _dynamic_threads,
    Priority _default_priority,
    CORBA::Boolean _allow_request_buffering = 0,
    CORBA::ULong _max_buffered_requests = 0,
    CORBA::ULong _max_request_buffer_size = 0 );
```

Creates a new Real-Time CORBA Threadpool with the specified configuration, and returns a `RTCORBA::ThreadpoolId` for it.

Parameter	Description
<code>_stacksize</code>	Stacksize, in bytes, for each thread in the Threadpool.
<code>_static_threads</code>	Number of threads to create at the time of Threadpool creation. This value can be zero, as long as <code>_dynamic_threads</code> is non-zero.
<code>_dynamic_threads</code>	Number of extra threads that can be created, if all the statically created threads are in use and more threads are required. This value can be zero (so that no more threads can be dynamically created), as long as <code>_static_threads</code> is non-zero.
<code>_allow_request_buffering</code>	Boolean flag to enable request buffering when all threads are in use. Not supported by VisiBroker for C++. The value of this parameter is ignored.
<code>_max_buffered_requests</code>	Maximum number of requests to buffer when all threads are in use. Not supported by VisiBroker for C++. The value of this parameter is ignored.
<code>_max_request_buffer_size</code>	Maximum amount of data to buffer, in bytes, when all threads are in use. Not supported by VisiBroker for C++. The value of this parameter is ignored.

```
void destroy_threadpool( ThreadpoolId _threadpool );
```

Destroys a Real-Time CORBA Threadpool. The Threadpool must not be in use by any Object Adapter, or the operation will fail, and a CORBA system exception is raised.

Parameter	Description
<code>_threadpool</code>	The ThreadpoolId of the Threadpool to destroy.

```
void threadpool_idle_time(
    ThreadpoolId _threadpool,
    CORBA::ULong _seconds );
```

Sets the time, in seconds, that dynamically allocated threads remain idle before they are garbage collected. Configured on a per-Threadpool basis. The default is to garbage collect dynamically allocated threads after 300 seconds.

This method is a proprietary VisiBroker extension.

Parameter	Description
<code>_threadpool</code>	The ThreadpoolId of the Threadpool to set the Idle Time for.
<code>_seconds</code>	The maximum number of seconds that a dynamically allocated thread can be idle in this Threadpool before it is destroyed. Statically allocated threads are not destroyed.

```
PriorityModelPolicy create_priority_model_policy(
    in PriorityModel _priority_model,
    in Priority _server_priority );
```

Creates an instance of the `RTCORBA::PriorityModelPolicy` policy object, for use in configuring one or more Real-Time POAs. See “[RTCORBA::PriorityModel](#)” on page 297 and “[RTCORBA::PriorityModelPolicy](#)” on page 298.

Parameter	Description
<code>_priority_model</code>	<code>RTCORBA::SERVER_DECLARED</code> , for the Server Declared Priority Model or <code>RTCORBA::CLIENT_PROPAGATED</code> for the Client Priority Propagation Model.
<code>_server_priority</code>	In the Server Model, the Real-Time CORBA Priority that invocations on objects activated on this POA will be executed at, provided a Priority value is not associated with the individual object at the time of activation. In the Client Model, the Real-Time CORBA Priority that invocations on objects activated on this POA will be executed at if they come from a non-Real-Time CORBA client or a Real-Time CORBA client that has not specified a Real-Time CORBA Priority on <code>RTCORBA::Current</code> before making the invocation.

```
ThreadpoolPolicy create_threadpool_policy(
    in ThreadpoolId _threadpool );
```

Creates an instance of the `RTCORBA::ThreadpoolPolicy` policy object, for use in configuring one or more Real-Time POAs.

Parameter	Description
<code>_threadpool</code>	The ThreadpoolId of the Threadpool to associate POA with.

RTCORBA::ThreadpoolId

```
typedef CORBA::ULong RTCORBA::ThreadpoolId
```

Values of the type `RTCORBA::ThreadpoolId` are used to identify Real-Time CORBA Thread-pools. A value of this type is returned from the `create_threadpool` method of `RTCORBA::RTORB`.

The ID can be used to initialize an instance of a Threadpool Policy, which in turn can be passed in to a call to `create_POA`, as a member of the `PolicyList` parameter, to configure a Real-Time POA. For details, see “[RTCORBA::RTORB](#)” on page 298, “[RTCORBA::ThreadpoolPolicy](#)” on page 302, and the section “Association of an Object Adapter with a Threadpool” in the *VisiBroker for C++ Developer's Guide*.

IDL definition

```
typedef unsigned long ThreadpoolId;
```

RTCORBA::ThreadpoolPolicy

```
class RTCORBA::ThreadpoolPolicy : CORBA::Policy
```

An instance of this Real-Time Policy type is created by calling the `create_threadpool_policy` method of `RTCORBA::RTORB`. The Policy instance can then be used to configure a Real-Time POA at the time of its creation, by passing it into the `create_POA` method, as a member of the Policy List parameter. See [“RTCORBA::RTORB” on page 298](#), [“RTCORBA::ThreadpoolId” on page 301](#), and the section “Association of an Object Adapter with a Threadpool” in the *VisiBroker for C++ Developer's Guide* for more information.

IDL definition

```
interface ThreadpoolPolicy : CORBA::Policy {  
    readonly attribute ThreadpoolId threadpool;  
};
```

Chapter 20

Pluggable Transport Interface Classes

This chapter describes the classes of the Pluggable Transport Interface provided by VisiBroker for C++. For information on how to implement support for a transport protocol via the VisiBroker Pluggable Transport Interface, see the chapter “VisiBroker Pluggable Transport Interface” in the Developer Guide.

Important For documentation updates, go to www.borland.com/techpubs/bes.

VISPTransConnection

This class is the abstract base class for a connection class that must be implemented for each transport protocol that is to be plugged in to VisiBroker, to allow VisiBroker to work with that particular transport protocol. Each instance of the derived class will represent a single connection between a server and a client. VisiBroker will request instances of this class be created (via the corresponding factory class, see “[virtual CORBA::Boolean waitNextMessage\(CORBA::ULong _timeout\) = 0;](#)”) on both the client and server side of the ORB, whenever a new connection is required.

Include file

The `vptrans.h` file should be included to use this class.

VISPTransConnection methods

```
virtual void close() = 0;
```

To be implemented by the derived connection class. This method closes the connection in an orderly fashion. This method must be able to close the connection from either the client- or the server-side of a connection.

```
virtual void connect(CORBA::ULongLong _timeout) = 0;
```

To be implemented by the derived connection class. This method will be called by the client-side ORB, and must communicate with the remote peer's 'Listener' instance to setup a new connection on the server-side. The function does not return any error code, but should throw exceptions if any transport layer errors occur. Any exception may be thrown, including a CORBA User Exception, as the exception will be thrown back to the client CORBA application. CORBA::TRANSIENT is one possible exception that could be thrown.

The timeout value is in specified in milliseconds. A value of 0 means no timeout (block forever), and this is the default value, which is used unless the timeout is set through the VisiBroker policy system. If the transport does not support timeouts on connect, it still can be used successfully. In this case the connect call must always block until the connection is established or has failed.

Parameter	Description
<code>_timeout</code>	Timeout value to use, in milliseconds. 0 indicates no timeout (block forever)

```
virtual void flush() = 0;
```

To be implemented by the derived connection class. If this transport buffers data, this method should immediately send all data buffered for output, and block until the data is sent. Otherwise, there is nothing to be done and it can return immediately.

```
virtual IOP::ProfileValue_ptr getPeerProfile() = 0;
```

To be implemented by the derived connection class. This method should return a copy of the Profile describing the peer endpoint used in this connection. The copy must be created on the heap and the caller is responsible for releasing the used memory. The Profile does not describe the actual connection for this instance, but the Profile of the 'Listener' endpoint used during the 'connect' call.

```
virtual CORBA::Long id() = 0;
```

To be implemented by the derived connection class. This method must return a unique number for each connection instance. The ID only needs to be unique for this transport. It is used to lookup/locate a connection instance during request dispatching for this transport.

```
virtual CORBA::Boolean isBridgeSignalling() = 0;
```

To be implemented by the derived connection class. This method is used to indicate to the ORB which worker thread 'cooling' strategy is to be used. If the method returns 0 (FALSE), it means that the protocol plug-in itself is going to handle the re-reading of the connection after a request has been read. This is only possible if the plug-in is capable of doing a blocking read with timeout on the protocol endpoint. If it cannot or chooses not to, this method should return 1 (TRUE), and the transport bridge will notify the thread if another request becomes available or the when the timeout is reached. Note that thread cooling only occurs if a cooling time is configured for that protocol instance.

```
virtual CORBA::Boolean isConnected() = 0;
```

To be implemented by the derived connection class. This method should return 1 (TRUE), if the remote peer is still connected. If the connection was closed by the peer or any error condition exists that prevents the use of this connection, it must return 0 (FALSE).


```
virtual CORBA::Boolean isDataAvailable() = 0;
```

To be implemented by the derived connection class. This method should return 1 (TRUE), if data is ready to be read from the connection. Otherwise, it should return 0 (FALSE).

```
virtual CORBA::Boolean no_callback() = 0;
```

To be implemented by the derived connection class. This method indicates whether a connection of this transport can be used to reverse the client/server setup and call back to a servant in the client code. It should return 0 (FALSE) if it can not, which will cause the ORB to create a new connection for this kind of call, or 1 (TRUE) if it can.

This feature is provided to support Bi-Directional IIOp, that was introduced in GIOP-1.2. See the CORBA specification for details.

```
virtual void read(CORBA::Boolean _isFirst, CORBA::Boolean _isLast, char* _data,  
CORBA::ULong _offset, CORBA::ULong _length, CORBA::ULongLong _timeout)= 0;
```

To be implemented by the derived connection class. This method reads data from the connection. It does not return any error code, but must signal transport related errors by throwing exceptions. The arguments describe a byte array with a given length that needs to be filled. This function must either fill the complete byte array successfully, timeout, or throw an exception.

The timeout parameter's value defaults to 0 unless the user sets it through the VisiBroker QoS policies. A value of 0 indicates no timeout, and hence that the read should block forever waiting for data. Therefore, if this transport does not support timeouts on read/write, it still can be used successfully. In this case the read call must always block until all data has arrived.

Parameter	Description
<code>_isFirst</code>	TRUE if this is the first time data is being read from the connection.
<code>_isLast</code>	TRUE if this is the last time data is being read from the connection.
<code>_data</code>	Byte array to read data into.
<code>_offset</code>	Offset into the array at which to start storing the read data.
<code>_length</code>	The number of bytes of data to be read.
<code>_timeout</code>	Timeout value to use, in milliseconds. 0 indicates no timeout (block forever)

```
virtual void setupProfile(const char* prefix, VISPTransProfileBase_ptr peer) = 0;
```

To be implemented by the derived connection class. This method is used to tell a newly created client-side connection object what peer it should try to connect to in later steps. (When `connect()` is called.) The given `VISPTransProfileBase_ptr` base class should be cast to the Profile class type of the particular transport and all member data in the connection should be initialized from that instance. A prefix string is also passed, for property lookup, in case additional property parameters need to be read.

Parameter	Description
<code>prefix</code>	String prefix of the form "vbroker.se.<SE_name>.scm.<SCM_name>" that the method can use to read any protocol-specific VisiBroker properties that may have been set to configure this instance.
<code>peer</code>	Profile for the Listening endpoint that this connection will connect to. Given as an instance of this protocol's Profile class, passed as a pointer to the base <code>VISPTransProfile</code> class.

```
virtual CORBA::Boolean waitNextMessage(CORBA::ULong _timeout) = 0;
```

To be implemented by the derived connection class. This method should block the calling thread until either data has arrived on this connection or the given timeout (in milliseconds) has expired. It should return 1 (TRUE) if data is available, or 0 (FALSE) if not. Note that a value of 0 for the `_timeout` parameter should never occur (as in this case the ORB should not call this method). Therefore receiving this value should be handled as an error, perhaps by logging an error message.

Parameter	Description
<code>_timeout</code>	Maximum amount of time to wait for a message (in seconds). 0 means wait forever.

```
virtual void write(CORBA::Boolean _isFirst, CORBA::Boolean _isLast, char* _data,
CORBA::ULong _offset, CORBA::ULong _length, CORBA::ULongLong _timeout) = 0;
```

To be implemented by the derived connection class. This method sends data through the connection to the remote peer. It does not return any error code, but must signal transport related errors by throwing exceptions. The arguments describe a byte array with a given length that needs to be sent. This function must either send the complete byte array successfully, timeout, or throw an exception. The timeout parameter's value defaults to 0 unless the user sets it through the VisiBroker QoS policies. A value of 0 indicates no timeout, and hence that the write should block forever waiting for data. Therefore, if this transport does not support timeouts on read/write, it still can be used successfully. In this case the write call must always block until all data has arrived.

Parameter	Description
<code>_isFirst</code>	TRUE if this is the first time data is being sent through the connection.
<code>_isLast</code>	TRUE if this is the last time data is being sent through the connection.
<code>_data</code>	Byte array of data that needs to be sent.
<code>_offset</code>	Offset into the array at which to start storing the read data.
<code>_length</code>	The number of bytes of data to be sent.
<code>_timeout</code>	Timeout value to use, in seconds. 0 indicates no timeout (block forever)

VISPTransConnectionFactory

This class is the abstract base class for a connection factory class that must be implemented for each transport protocol that is to be plugged in to VisiBroker, to allow VisiBroker to work with that particular transport protocol. A singleton instance of the derived class is registered with VisiBroker, via the `VISPTransRegistrar` class, described later. The ORB calls the connection factory object to create instances of the connection class of the associated transport. The connection class is the corresponding class derived from class `VISPTransConnection`.

Include file

The `vptrans.h` file should be included to use this class.

VISPTransConnectionFactory methods

```
VISPTransConnection_ptr create(const char* prefix) = 0;
```

To be implemented by the derived connection factory class. This method creates a new instance of the corresponding connection class and returns the pointer to it cast to the base class type. The caller is responsible for the destruction of the instance when it is no longer required.

Parameter	Description
prefix	String prefix of the form “vbroker.se.<SE_name>.scm.<SCM_name>” that the method can use to read any protocol-specific VisiBroker properties that may have been set to configure the connection factory.

VISPTransListener

This class is the abstract base class for a listener factory that must be implemented for each transport protocol that is to be plugged in to VisiBroker, to allow VisiBroker to work with that particular transport protocol. Instances of the derived class are created each time a Server Engine is created that includes Server Connection Managers (‘SCMs’) that specify the particular transport protocol. One instance is created per SCM instance that specifies the protocol. The listener instances are used by the server-side ORB to wait for incoming connections and requests from clients. New connections and requests on existing connections are signalled by the listener to the ORB via the Pluggable Transport Interface’s Bridge class (see “[VISPTransBridge](#)” on page 312). When a request is received on an existing connection, the connection goes through a ‘Dispatch Cycle’. The Dispatch Cycle starts when the connection delivers data to the transport layer. In this initial state, the arrival of this data must be signalled to the ORB via the Bridge and then the Listener ignores the connection until the Dispatch process is completed (in the mean time, the connection is said to be in the ‘dispatch state’). The connection is returned to the initial state when the ORB makes a call to the Listener’s `completedData()` method. During the dispatch state the ORB will read directly from the connection until all requests are exhausted, avoiding any overhead incurred by the Bridge-Listener communication. In most cases, the transport layer uses blocking calls that wait for new connections. In order to handle this situation, the Listener should be made a subclass of the class `VISThread` and start a separate thread of execution that can be blocked without holding up the whole ORB.

Include file

The `vptrans.h` file should be included to use this class.

VISPTransListener methods

```
virtual void completedData(CORBA::Long id) = 0;
```

To be implemented by the derived listener class. This method is called when the ORB has completed reading a request from the connection with the given id and wants the Listener once again to signal any new incoming requests on that connection (via the Bridge).

Parameter	Description
id	Id of the connection that may once again be listened on.

```
virtual void destroy() = 0;
```

To be implemented by the derived listener class. This method instructs the Listener instance to tear down its endpoint and close all related active connections.

```
virtual IOP::ProfileValue_ptr getListenerProfile() = 0;
```

To be implemented by the derived listener class. This method should return the Profile describing the Listener instance's endpoint on this transport. The returned Profile should be a copy on the heap and the caller (the ORB) takes over memory management of it.

```
virtual CORBA::Boolean isDataAvailable(CORBA::Long id) = 0;
```

To be implemented by the derived connection factory class. This method should return 1 (TRUE), if the connection with the given Id has data ready to be read. Returns 0 (FALSE) otherwise. Normally the call should just be forwarded to the transport layer to find out.

Parameter	Description
id	Id of the connection that should be queried to see if data is available.

```
virtual void setBridge(VISPTransBridge* up) = 0;
```

To be implemented by the derived listener class. This method establishes the 'link' to the Pluggable Transport Bridge instance to be used by this Listener instance. The pointer it passes to the Listener should be stored to allow 'upcalls' to be made into ORB when necessary.

Parameter	Description
up	Pointer to Pluggable Transport Bridge instance that the Listener instance should use to communicate with the ORB.

VISPTransListenerFactory

This class is the abstract base class for a listener factory class that must be implemented for each transport protocol that is to be plugged in to VisiBroker, to allow VisiBroker to work with that particular transport protocol. A singleton instance of the derived class is registered with VisiBroker, via the VISPTransRegistrar class. The ORB calls this object to create instances of the listener class of the associated transport. The listener class is the corresponding class derived from class VISPTransListener, as described in "[VISPTransListener](#)".

Include file

The **vptrans.h** file should be included to use this class.

VISPTransListenerFactory methods

```
VISPTransListener_ptr create(const char* propPrefix) = 0;
```

To be implemented by the derived listener factory class. This method creates a new instance of the corresponding listener class and returns the pointer to it cast to the base class type. The caller (the ORB) is responsible for the destruction of the instance when it is no longer required.

Parameter	Description
propPrefix	String prefix of the form “vbroker.se.<SE_name>.scm.<SCM_name>” that the method can use to read any protocol-specific VisiBroker properties that may have been set to configure the listener instance or the particular listener instance that is being created. Note that the factory can pass the prefix into the constructor of the listener instance it is creating, to allow it to read properties itself. This would require the derived listener class to have a constructor that takes the prefix as a parameter.

VISPTransProfileBase

```
class VISPTransProfileBase : public GIOP::ProfileBodyValue, public
CORBA_DefaultValueRefCountBase
```

This class is the abstract base class for a Profile class that must be implemented for each transport protocol that is to be plugged in to VisiBroker, to allow VisiBroker to work with that particular transport protocol. This class provides the functionality to convert between a transport specific endpoint description and an CORBA IOP based IOR that can be exchanged with other CORBA implementations. It is also used during the process of binding a client to a server, by passing a ProfileValue to a ‘parsing’ function that has to return TRUE or FALSE, to determine whether a particular IOR is usable for this transport or not. An instance of the derived Profile class is frequently passed to functions via a pointer to its base class type. In order to support safe runtime downcasting with any C++ compiler, a ‘_downcast’ function must be provided that can test if the cast is legal or not.

Include file

The **vptrans.h** file should be included to use this class.

VISPTransProfileBase methods

```
static GIOP::ObjectKey* convert(const PortableServer::ObjectId& seq);
```

Converts octet sequence representation of an Object Key into the in-memory representation.

Parameter	Description
seq	Octet sequence version of Object Key, to be converted into in-memory representation.

```
void object_key(GIOP::ObjectKey_ptr k);
```

Set the Object Key for this Profile instance.

Parameter	Description
k	Object key

```
const GIOP::ObjectKey_ptr object_key() const;
```

Get the Object Key for this Profile instance.

```
void version(const GIOP::Version& v);
```

Set the GIOP version for this Profile.

Parameter	Description
v	GIOP Version

```
GIOP::Version& version();
```

Get the GIOP version of this Profile.

```
const GIOP::Version& version() const;
```

Get the GIOP version of this Profile.

```
static const VISValueInfo& _info();
```

Get the VisiBroker ValueInfo for this Profile type.

VISPTransProfileBase members

```
static const VISValueInfo& _stat_info;
```

Stores the VisiBroker ValueInfo for this particular Profile type.

VISPTransProfileBase base class methods

```
IOP::ProfileValue_ptr copy()
```

To be implemented by the derived listener factory class. This method should make an exact copy on the free store and return a pointer to it. It is good coding practice to use the copy constructor inside of this function.

```
CORBA::Boolean matchesTemplate(IOP::ProfileValue_ptr body);
```

To be implemented by the derived Profile class. This method should return 1 (TRUE) if there is an IOR in the given data, that can be used to connect through this transport. Otherwise return 0 (FALSE).

Parameter	Description
body	body Profile to be checked, to see if it can be used by this transport.

```
IOP::ProfileId tag();
```

To be implemented by the derived Profile class. This method should return the unique tag value for this Profile.

```
IOP::TaggedProfile* toTaggedProfile();
```

To be implemented by the derived Profile class. This method should return a tagged (stringified) Profile instance created with the values read from this instance's member data.

```
static VISPTransProfileBase* _downcast(CORBA::ValueBase* vbptr);
```

To be implemented by the derived Profile class. Function to downcast a base class pointer to an instance of this Profile class.

Parameter	Description
vbptr	Profile instance passed as base Value type pointer.

```
virtual void* _safe_downcast(const VISValueInfo &info) const;
```

To be implemented by the derived listener factory class. Virtual method called by ORB during downcast, to check type info data.

Parameter	Description
info	VisiBroker Value Info for this Profile type.

VISPTransProfileFactory

This class is the abstract base class for a Profile factory class that must be implemented for each transport protocol that is to be plugged in to VisiBroker, to allow VisiBroker to work with that particular transport protocol. A singleton instance of the derived class is registered with VisiBroker, via the VISPTransRegistrar class. The ORB calls this object to create instances of the Profile class of the associated transport. The Profile class is the corresponding class derived from class VISPTransProfileBase, as described in “[VISPTransProfileBase](#)”.

Include file

The `vptrans.h` file should be included to use this class.

VISPTransProfileFactory methods

```
IOP::ProfileValue_ptr create(const IOP::TaggedProfile& profile)
```

Read the tagged IOR and create a Profile describing a Listener endpoint.

Parameter	Description
profile	CDR encoded IOR to be read.

```
CORBA::ULong hash(VISPTransProfileBase_ptr prof);
```

Support the optimized storage of profiles in a hashed lookup table by calculating a hash number for the given instance. Return 0 if you do not provide hash values.

Parameter	Description
prof	Profile instance to produce hash value for.

```
IOP::ProfileId getTag();
```

Return the unique Profile Id tag for the type of Profile created by this factory.

VISPTransBridge

This class provides a generic interface between the transport classes and the ORB. It provides methods to signal various events occurring in the transport layer.

Include file

The `vptrans.h` file should be included to use this class.

VISPTransBridge methods

```
CORBA::Boolean addInput(VISPTransConnection_ptr con);
```

Send a connection request to the ORB through the bridge, by passing a pointer to the Connection instance representing the Listener endpoint. The returned flag signals whether the ORB has accepted the new connection (returns 1 (TRUE)) or refused it (returns 0 (FALSE)). The latter might happen due to resource constraints or due to a restriction on connections (set up through the property system).

Members	Description
con	Connection object representing the Listener endpoint wish to connect to.

```
void signalDataAvailable(CORBA::Long conId);
```

Passes the connection id to the ORB of a connection that just got new data from the transport layer. This will start the dispatch cycle for incoming requests.

Members	Description
conId	Connection Id of connection want to indicate data is available on

```
void closedByPeer(CORBA::Long conId);
```

Tell the ORB that the connection with the given id was closed by the remote peer.

Members	Description
conId	Connection Id of connection want to indicate was closed by the remote peer.

VISPTransRegistrar

This class must be used to register a new transport with the ORB. The protocol name string given during registration is used as identifier of this transport and must be unique in the scope of that ORB. It is also used as a prefix in the name string of properties related to this transport.

Include file

The `vptrans.h` file should be included to use this class.

VISPTransRegistrar methods

```
static void addTransport(const char* protocolName, VISPTransConnectionFactory*
connFac, VISPTransListenerFactory* listFac, VISPTransProfileFactory* profFac);
```

Register the protocol name string and the three Factory instances used to create specific classes for this transport. This method is static and can therefore be called at any time during the initialization of the ORB.

Members	Description
<code>protocolName</code>	Name to be used to identify this transport protocol.
<code>connFac</code>	Pointer to singleton instance of connection factory.
<code>listFac</code>	Pointer to singleton instance of Listener factory.
<code>profFac</code>	Pointer to singleton instance of Profile factory.

VisiBroker for C++ Logging

This section describes the classes that support VisiBroker for C++ logging.

VISDLoggerMgr

This class is a bootstrap class into the functionality provided by the logging library `vdlog`.

Include file

Include the `vdlog.h` file when you use this class.

VISDLoggerMgr methods

```
static VISDLoggerMgr_ptr instance();
```

Static function to access a singleton instance of `VISDLoggerMgr`.

```
CORBA::Boolean global_log_enabled();
```

Returns true if the global log switch is enabled, else false.

```
void global_log_enabled(CORBA::Boolean b);
```

Setter method for the global log level switch.

Parameter	Description
<code>b</code>	boolean value to enable or disable the global log level switch

```
VISDLogLevel::Level global_log_level();
```

Returns the current global log level (verbosity) setting on the log manager.

```
void global_log_level(VISDLogLevel::Level l);
```

Setter for the global log level on the log manager.

Parameter	Description
l	global verbosity setting

```
VISDLogger_ptr get_default_logger();
```

Returns the default logger. If not created, creates and returns. The name of the returned logger is “default”.

```
VISDLogger_ptr get_logger(const char* name, VISDAppender_ptr* apps = NULL,
CORBA::Short num_apps = 0);
```

Creates if not created and returns a logger with the given name.

Parameter	Description
name	input name of the logger
apps	pointer to an array of appender pointers indicating an initial list of appender for the logger
num_apps	number of appenders in the array of appender pointer

```
void register_app_factory(VISDAppenderFactory* fac);
```

API for custom appender factories to register themselves with the logger framework. Factory will be added to a dictionary of appender factories indexed by its name. If a factory is not registered with the framework, then an instance of its type cannot be created.

Parameter	Description
fac	appender factory to be registered

```
VISDAppender_ptr create_app(const char* logger_name,
VISDConfig::LogAppenderConfig_ptr p);
```

API to create an appender for the logger specified by its name using the configuration information pointed.

Parameter	Description
logger_name	name of the logger for which the appender instance is to be created
p	pointer to the logger appender instance configuration

```
void register_lyt_factory(VISDLayoutFactory* fact);
```

API for custom layout factories to register themselves with the logger framework.

Parameter	Description
fact	pointer to the implemented layout factory to be registered

```
VISDLayout_ptr create_lyt(const char* logger_name,
VISDConfig::LogAppenderConfig_ptr p);
```

API to create a layout instance.

Parameter	Description
logger_name	name of the logger in which the appender instance is associated which needs to use the layout
p	pointer to the logger appender instance configuration

VISDLogger

Class providing the logging interface.

Include file

Include the **vdlog.h** file when you use this class.

VISDLogger methods

```
const char* name() const;
```

Returns the name of the logger object.

```
void log(VISDLogLevel::Level level, const char* message, const char* sourcefile =
NULL, CORBA::ULong linenum = 0, const void* bindata = NULL, size_t binsize = 0);
```

API to log messages.

Parameter	Description
level	log level of the logged message
message	logged message data
sourcefile	source file name from where the message is being logged
linenum	source file line number from where the message is being logged
bindata	binary data pointer
binsize	size of any binary data

```
void log(VISDLogLevel::Level level, const char* component, const char* message,
const char *sourcefile = NULL, CORBA::ULong linenum = 0, const void *bindata =
NULL, size_t binsize = 0)
```

API to log messages.

Parameter	Description
level	log level of the logged message
component	source name from where the message is being logged. The source name is the logical module name that can be useful during filtering
message	logged message data
sourcefile	source file name from where the message is being logged
linenum	source file line number from where the message is being logged
bindata	binary data pointer
binsize	size of any binary data

VISDAppenderFactory

Interface for appender factory implementations to implement. The logger framework calls on this interface for appender instance creation.

Include file

Include the **vdlog.h** file when you use this class.

VISDAppenderFactory methods

```
virtual const char* type_name() = 0;
```

This method is invoked by the logger framework when it needs to know the type of the factory. For example, when a factory registers itself with the logger manager, this API is called to get the type name. The type name identifies the type of destination to which its appenders will forward the logger. Type names “stdout”, “rolling” and others as mentioned in the developer guide are restricted from usage. Should return back a unique type name for the appender type.

```
virtual VISDAppender_ptr create(const char* logger_name,  
VISDConfig::LogAppenderConfig_ptr p) = 0;
```

This method is invoked by the logger framework when it needs to create an instance of the appender supported by this factory. The return value should be an instance of desired appender.

Parameter	Description
logger_name	name of the logger on which the appender instance is to be associated with
p	pointer to the logger's appender instance configuration.

```
virtual void destroy(VISDAppender_ptr p) = 0;
```

This method is invoked by the logger framework when it is done with using the appender instance. The API is supposed to remove all resources dedicated to the appender instance when it was created.

Parameter	Description
p	Appender instance pointer that is to be destroyed

VISDAppender

```
class VISDAppender : public VISResource
```

Interface providing the appender interface. The logger object uses this interface to log to specific destinations..

Include file

Include the **vdlog.h** file when you use this class.

VISDAppender methods

```
virtual VISDAppenderFactory* factory() = 0;
```

Should return the associated factory object which created this appender instance.

```
virtual CORBA::Boolean append(const VISDLogRecord& record) = 0;
```

API used by the logger to forward the log message to a specific destination. The log record abstracts the complete log message. On successful completion of forwarding, the API should return TRUE..

Parameter	Description
record	log record to be appended to the destination

```
virtual CORBA::Boolean ORB_initialized(void* orb_ptr) = 0;
```

This is a notification from the ORB that it has initialized. If an appender is going to use any of the ORB functionality, then it needs to wait for this notification and return back TRUE. Otherwise, it should return back FALSE. After this notification, the appender can start using any of the ORB interfaces.

Parameter	Description
orb_ptr	reference to the ORB

```
virtual void ORB_shutdown() = 0;
```

This is a notification from the ORB that it is shutting down. If the appender is using any ORB functionality, then it needs to stop using that after this notification.

VISDLayoutFactory

Interface for layout factory implementations to implement. The logger framework calls on this interface for layout instance creation.

Include file

Include the **vdlog.h** file when you use this class.

VISDLayoutFactory methods

```
virtual const char* type_name() = 0;
```

Returns the type name of the layout that this factory will create.

```
virtual VISDLayout_ptr create(const char* logger_name,  
VISDConfig::LogAppenderConfig_ptr p) = 0;
```

Should creates a layout instance. This API is called by the logger framework when an instance of the layout is desired.

Parameter	Description
logger_name	name of the logger whose associated appender instance needs this layout instance
p	pointer to the logger's appender instance configuration.

```
virtual void destroy(VISDLayout_ptr layout) = 0;
```

Framework calls this API when it is done with usage of the layout and needs to factory to destroy the instance.

Parameter	Description
layout	pointer to layout instance which needs to be destroyed

VISDLayout

```
class VISDLayout : public VISResource
```

Interface which all layout instances should implement. Appenders which desire to format the log message before outputting to the desired destination will make use of this interface.

Include file

Include the **vdlog.h** file when you use this class.

VISDLayout methods

```
virtual VISDLayoutFactory* factory() = 0;
```

Should return the factory of the layout instance that created it.

```
virtual void format(const VISDLogRecord& record, char* buf, CORBA::ULong buf_size, CORBA::String_var& other_buf) = 0;
```

API that is called by the appender instances for formatting the log record. The appender allocates buffer and sends the buffer into this API and expects the layout to format the message and set in this buffer. However, if the layout wants more memory than that has been sent to it by the appender, then it can itself allocate memory and make use of `other_buf`.

Parameter	Description
record	log record containing the log message
buf	memory buffer sent by the appender onto which the layout can set the formatted message
buf_size	size of the memory buffer sent in by the appender
other_buffer	If the layout needs more memory than that sent by the appender, then it can allocate memory into this buffer and set the formatted text into it.

VISDConfig

Namespace class for configuration structures.

Include file

Include the **vdlog.h** file when you use this class.

LogAppenderConfig structure

```
struct LogAppenderConfig {
    CORBA::String_var appender_name;
    CORBA::String_var appender_type;
    CORBA::String_var layout_type;
};
typedef LogAppenderConfig* LogAppenderConfig_ptr;
```

This structure contains a single appender instance configuration on a logger. This is filled and passed to the factory interfaces by the logger framework after reading from the configurations.

Members	Description
appender_name	Name of the appender instance configured on the logger
appender_type	Type name of the appender instance. This implies which appender factory needs to be used
layout_type	Type name of the layout instance desired. Again this implies which layout factory should be used to obtain layout instance.

VISDLogRecord

Class abstracting a log message. Apart from the actual log message, it also captures various other states such as thread id, timestamp etc.parameter.

Include file

Include the **vdlog.h** file when you use this class.

VISDLogRecord methods

```
Timestamp get_timestamp() const;
```

Returns the timestamp of the log record.

```
CORBA::ULong get_seq_number() const;
```

Returns a sequence number if many log records are logged at the same time interval.

```
CORBA::ULong get_process_id() const;
```

Returns the process id.

```
CORBA::ULong get_thread_id() const;
```

Returns the thread id of the thread that logged this message.

```
const char* get_thread_name() const;
```

If the thread is named, then it returns the thread name.

```
const char* get_logger_name() const;
```

Returns the logger object's name.

```
VISDLogLevel::Level get_log_level() const;
```

Returns the verbosity of the logged message.

```
const char* get_component_name() const;
```

Returns the source name of the source that logged the message.

```
const char* get_filename() const;
```

Returns the name of the file that logged the message.

```
CORBA::ULong get_line_number() const;
```

Returns the line number in the file from where the log message is emanating.

```
const char* get_message() const;
```

This is the actual logged message.

```
const unsigned char* get_bindata() const;
```

Returns any binary data that is piggybacking on the log record.

```
size_t get_binsize() const;
```

Returns the size of the binary data.

VISDLogLevel

Class enclosing verbosity enumeration Level.

Include file

Include the `vdlog.h` file when you use this class.

Level enumeration

```
enum Level {
    OFF_      = 1000,
    EMERG_    = 800,
    EXCEP_    = 800,
    FATAL_    = 800,
    ALERT_    = 700,
    CRIT_     = 600,
    ERROR_    = 500,
    WARN_     = 400,
    NOTICE_  = 300,
    INFO_     = 200,
    DEBUG_    = 100,
    ALL_      = 0,
    DEFAULT_  = -1
};
```

Index

Symbols

[] brackets 4
| vertical bar 4
... ellipsis 4

Numerics

5.x interceptors
 interceptor managers 242
 InterceptorManager class 242
 IOR templates 242

A

accessing
 system exceptions 62
 the interface repository 105
 user exceptions 62
ActiveObjectLifecycleInterceptor, class 248
ActiveObjectLifecycleInterceptorManager, class 249
Adapter activators 9
AdaptorActivator, methods 10
agent class 279
Agent methods 280
AliasDef 79
 class 79
 methods 79
all_repository_ids 280
Any 49, 65, 303, 315
 class 49, 303, 315
 extraction operators 51
 initialization operators 50
 methods 49, 303, 315
arguments
 -ORBid 236
 -ORBServerId 236
ArrayDef 80
 class 80
 methods 80
AttributeDef 80
 class 80
AttributeDescription 81
 class 81
AttributeMode 82
 class 82

B

BAD_INV_ORDER
 ClientRequestInfo 215
 Current methods 221
 ORBInitInfo 229
 ServerRequestInfo 235
BAD_PARAM
 ClientRequestInfo 215
 IORInfo 224
Basic Object Adaptor (see BOA) 11

bind options
 connection_timeout 10
 defer_bind 10
 enable_rebind 10
 max_bind_tries 10
 receive_timeout 10
 send_timeout 10
Binding 129
binding clients to objects 19
Binding structure 129
BindingIterator
 class 130
 methods 130
BindingList sequence 129
BindingList, class 129
BindInterceptor 243
BindInterceptorManager, class 244
BindOptions 10
 struct 10
BOA 11
 include file 115, 117, 120, 121
 methods 12
 VisiBroker extensions 15
Borland Developer Support, contacting 4
Borland Technical Support, contacting 4
Borland Web site 4, 5

C

C++ language exceptions 62
CancelRequestHeader 266
ChainUntypedObjectWrapperFactory 254
 class 254
class
 ActiveObjectLifecycleInterceptor 248
 ActiveObjectLifecycleInterceptorManager 249
 agent 279
 AliasDef 79
 Any 49, 65, 303, 315
 ArrayDef 80
 AttributeDef 80
 AttributeDescription 81
 AttributeMode 82
 Binding 129
 BindingIterator 130
 BindingList 129
 BindInterceptor 243
 BindInterceptorManager 244
 BOA 11
 ChainUntypedObjectWrapperFactory 254
 ClientInterceptor 245
 ClientRequestInceptorManager 246
 ClientRequestInfo 214
 ClientRequestInterceptor 216
 Codec 218
 CodecFactory 220
 CompletionStatus 16
 ConstantDef 82

- ConstantDescription 83
- Contained 83, 85, 99, 100
- Container 85, 99
- Context 16
- ContextList 51, 306, 317
- CORBA::PolicyManager 259
- Current 220, 291
- DuplicateName 228
- DynamicImplementation 53, 307, 318
- DynAny 53, 308, 318
- DynAnyFactory 57, 309, 319
- DynArray 57, 311, 320
- DynEnum 58, 312, 320
- DynSequence 59, 313, 321
- DynStruct 60, 322
- DynUnion 61
- EnumDef 91
- Environment 62
- Exception 19, 45
- ExceptionDef 92
- ExceptionList 63, 222
- ExtendedNamingContextFactory 132
- Fail 285
- FixedDef 93
- FormatMismatch 219
- ForwardRequest 222
- IDLType 95, 99
- Interceptor 223
- InterceptorManager 242
- InterceptorManagerControl 242
- InterfaceDef 96
- interface_name 7
- InvalidName 228
- InvalidTypeForEncoding 219
- IORCreationInterceptor 251
- IORInfo 223
- IORInfoExt 225
- IORInterceptor 226
- IRObjct 99
- MarshalInBuffer 271, 275
- MarshalOutBuffer 271, 275
- Messaging::RebindPolicy 262
- ModuleDef 99
- ModuleDescription 99
- Mutex 292
- NamedValue 65
- NamingContext 123
- NamingContextExt 128
- NamingContextFactory 131
- NativeDef 100
- NVList 16, 65, 66
- Object 19
- ObjectStatus 120
- ObjectStatusList 121
- ObjectWrapper 8
- OperationDef 100
- ORB 24
- ORBInitializer 227
- ORBInitInfo 228
- ParameterList 231
- _POA_ 8
- POA 30
- POALifeCycleInterceptor 247
- POALifeCycleInterceptorManager 248
- PolicyFactory 232
- PrimitiveDef 104
- PriorityMapping 295
- PriorityModelPolicy 297
- QoSExt::DeferBindPolicy 263
- Repository 105
- Request 69, 72, 75
- RequestInfo 232
- RTORB 298
- Seq 287
- SeqSeq 288
- SequenceDef 106
- ServantActivator 42
- ServantLocator 44
- ServantManager 45
- ServerRequestInfo 235
- ServerRequestInterceptor 238, 249
- ServerRequestInterceptorManager 251
- StringDef 107, 109
- StructDef 107
- SystemException 45
- ThreadpoolPolicy 301
- _tie_ 8
- TriggerHandler 286
- TypedDef 108
- TypeMismatch 219
- UnionDef 109
- UnknownEncoding 220
- UntypedObjectWrapper 256
- UntypedObjectWrapperFactory 257
- ValueBoxDef 110
- ValueDef 111
- _var 8
- VISClosure 253
- VISClosureData 254
- VISInit 289
- WstringDef 114
- class CORBA, Object 260
- classes 213
- ClientInterceptor 245
- ClientRequestInfo
 - BAD_INV_ORDER 215
 - BAD_PARAM 215
 - class 214
 - exceptions 215
 - INV_POLICY 215
 - methods 215
- ClientRequestInterceptor
 - class 216
 - exceptions 217
 - ForwardRequest 217
 - methods 217
- ClientRequestInterceptorManager, class 246
- Codec
 - class 218
 - exceptions 219
 - FormatMismatch 219
 - InvalidTypeForEncoding 219
 - members 219
 - methods 219
- Codec encoding, struct 221
- CodecFactory
 - class 220
 - exceptions 220
 - UnknownEncoding 220
- commands, conventions 4
- COMPLETED_MAYBE 16
- COMPLETED_NO 16

- COMPLETED_YES 16
- CompletionStatus 16
- ConstantDef 82
 - class 82
- ConstantDescription, class 83
- ConsumerAdmin
 - interface 133
 - method 133
- Contained 83, 99, 100
 - methods 84
- Container 85, 99
 - methods 86
- containment hierarchy 85
- Context 16
 - class 16
 - include file 16
 - methods 16
- ContextList
 - class 51, 306, 317
- Context_var class 16
- CORBA::BOA methods 12
- creation
 - Current 292
 - Mutex 293
 - PriorityMapping 295
 - RTORB 298
- Current 291
 - class 18, 220, 291
 - methods 19, 221, 292
- Current methods
 - BAD_INV_ORDER 221
 - exceptions 221
 - InvalidSlot 221

D

- deactivating, object implementations 11
- DeferBindPolicy, class 263
- defining an ORB object's interface 96
- DefinitionKind 90
 - enum 90
- delegation implementations 8
- deriving Interface Repository objects 83
- Desc structure 284
- destruction
 - Current 292
 - Mutex 293
 - PriorityMapping 295
 - RTORB 298
- Developer Support, contacting 4
- documentation 2
 - .pdf format 3
 - accessing Help Topics 3
 - Borland Security Guide 2
 - on the web 5
 - platform conventions used in 4
 - type conventions used in 4
 - updates on the web 3
 - VisiBroker for .NET Developer's Guide 2
 - VisiBroker for C++ API Reference 2
 - VisiBroker for C++ Developer's Guide 2
 - VisiBroker for Java Developer's Guide 2
 - VisiBroker GateKeeper Guide 3
 - VisiBroker Installation Guide 2
 - VisiBroker VisiNotify Guide 2
 - VisiBroker VisiTelcoLog Guide 2

- VisiBroker VisiTime Guide 2
- VisiBroker VisiTransact Guide 2
- DuplicateName
 - class 228
 - ORBInitInfo 229
- dynamic interfaces 49, 303, 315
- DynamicImplementation 53, 307, 318
 - class 53, 307, 318
 - methods 53, 307, 318
- DynAny 53, 308, 318
 - class 53, 308, 318
 - methods 54, 309, 319
 - usage restrictions 54
- DynAnyFactory, class 57, 309, 319
- DynArray 57, 311, 320
 - class 57, 311, 320
 - methods 57, 311, 320
 - usage restrictions 57
- DynEnum 58, 312, 320
 - class 58, 312, 320
 - methods 58, 312, 321
 - usage restrictions 58, 61
- DynSequence 59, 313, 321
 - class 59, 313, 321
 - methods 59, 313, 321
 - usage restrictions 59
- DynStruct 60, 322
 - class 60, 322
 - methods 60, 322
 - usage restrictions 60
- DynUnion 61
 - class 61
 - methods 61

E

- encoding
 - members 222
 - struct 221
 - supported 221
- enum
 - DefinitionKind 90
 - OperationMode 103
 - ParameterMode 103
 - PrimitiveKind 104
 - PriorityModel 297
- EnumDef 91
 - class 91
- enumeration
 - AttributeMode 82
 - DefinitionKind 90
 - OperationMode 103
 - ParameterMode 103
 - PrimitiveKind 104
 - TCKind 74
- Environment 62
 - methods 62
- event handlers, interfaces 123, 133
- EventChannel
 - interface 134
 - methods 134
- EventChannelFactory
 - interface 135
 - methods 135
- exception 45
 - class 19

- ExceptionDef 92
- ExceptionDescription 92
 - structure 92
- ExceptionList class 63, 222
- exceptions
 - BAD_INV_ORDER 236
 - BAD_PARAM 233
 - CodecFactory 220
 - DuplicateName 228
 - FormatMismatch 219
 - ForwardRequest 222, 238
 - InvalidName 228
 - InvalidSlot 233, 236
 - InvalidTypeForEncoding 219
 - INV_POLICY 236
 - IORInfo 224
 - NO_RESOURCES 238
 - ORBInitInfo 229
 - TypeMismatch 219
 - UNKNOWN 236
- extended methods, BOA 15
- ExtendedNamingContextFactory
 - class 132
 - methods 132

F

- Fail class 285
- FixedDef class 93
- FormatMismatch
 - class 219
 - Codec 219
- ForwardRequest
 - class 222
 - ClientRequestInterceptor 217
 - exceptions 222
- FullInterfaceDescription 93
 - structure 93
- FullValueDescription structure 94

G

- generated classes 7
 - _sk_ 8
 - _st_ 7
 - _tie_ 8
 - _var 8
- GIOP structure
 - CancelRequestHeader 266
 - LocateReplyHeader 266
 - LocateRequestHeader 267
 - RequestHeader 268
- GIOP structure::ReplyHeader 267
- GIOP_c.hh 267
- GLOBAL_SCOPE 12

H

- Help Topics, accessing 3

I

- IDL, OAD 116
- IDLType 95, 99
 - include file 95
 - methods 95
- IIOp structure, ProfileBody 269

- ImplementationStatus struct 115
- include file
 - BOA 115, 117, 120, 121
 - Context 16
 - IDLType 95
- interception points
 - receive_exception 217
 - receive_other 217
 - receive_reply 217
 - receive_request 238
 - receive_request_service_contexts 238
 - send_exception 238
 - send_other 238
 - send_poll 217
 - send_reply 238
 - send_request 217
- Interceptor
 - class 223
 - methods 223
- interceptor_c.hh 251
- InterceptorManager class 242
- InterceptorManagerControl class 242
- interceptors
 - client request 216
 - IOR 226
 - server request 238
- Interface Repository, classes 79
- InterfaceDef 96
 - methods 97
- InterfaceDescription structure 98
- interface_name class 7
- interfaces
 - ConsumerAdmin 133
 - EventChannel 134
 - EventChannelFactory 135
 - OAD 116
 - ProxyPullConsumer 136
 - ProxyPullSupplier 136
 - ProxyPushConsumer 136
 - ProxyPushSupplier 137
 - PullConsumer 137
 - PullSupplier 138
 - PushConsumer 138
 - PushSupplier 139
 - SupplierAdmin 139
- Interoperable Object Reference (see IOR) 269
- InvalidName
 - class 228
 - ORBInitInfo 229
- InvalidSlot Current methods 221
- InvalidTypeForEncoding
 - class 219
 - Codec 219
- INV_POLICY
 - ClientRequestInfo 215
 - IORInfo 224
- IOP structure::TaggedProfile 270
- IOR 269
- IORCreationInterceptor class 251
- IORInfo
 - BAD_PARAM 224
 - class 223, 225
 - exceptions 224
 - INV_POLICY 224
 - methods 224
 - validity 223

IORInfo class 223
IORInfoExt
 class 225
 methods 226
IORInterceptor
 class 226
 methods 226
IObject (Interface Repository object) 99
IObject (Interface Repository objects) methods 99

L

LOCAL_SCOPE 12
LocateReplyHeader 266
LocateRequestHeader 267
Location Service
 agent 279
 Fail 285
 Seq 287
 SeqSeq 288
 TriggerDesc 285
 TriggerHandle 286

M

manipulating object references 19
MarshalInBuffer
 class 271, 275
 methods 272, 275
MarshalOutBuffer, methods 276, 278
members
 argument in Parameter 231
 format in encoding 222
 major_version in encoding 222
 minor_version in encoding 222
 mode in Parameter 231
memory management semantics 16
MessageHeader 265
methods
 adapter_id in ServerRequestInfo 236
 adapter_manager_state_changed in
 IORInterceptor 226
 adapter_name in ServerRequestInfo 236
 adapter_state_changed in IORInterceptor 226
 adapter_template in IORInfo 224
 add_client_request_interceptor in ORBInitInfo 229
 add_ior_component in IORInfo 224
 add_ior_component_to_profile in IORInfo 224
 add_ior_interceptor in ORBInitInfo 229
 add_reply_service_context in
 ServerRequestInfo 236
 add_request_service_context in
 ClientRequestInfo 215
 add_server_request_interceptor in IORInfoExt 226
 add_server_request_interceptor in ORBInitInfo 229
 allocate_slot_id in ORBInitInfo 229
 arguments in ORBInitInfo 229
 arguments in RequestInfo 233
 bind in NamingContext 124
 bind_context in NamingContext 124
 bind_new_context in NamingContext 124
 BOA 12
 change_implementation in OAD 117
 codec_factory in ORBInitInfo 229
 components_established in IORInterceptor 226
 connect_push_supplier in ProxyPushConsumer 136

Contained 84
Container 86
Context 16
contexts in RequestInfo 233
create in EventChannelFactory 135
create_by_name in EventChannelFactory 135
create_codec in CodecFactory 220
create_policy in PolicyFactory 232
create_struct in Container 86
current_factory in IORInfo 224
decode in Codec 219
decode_value in Codec 219
destroy in EventChannel 134
destroy in EventChannelFactory 135
destroy in Interceptor 223
destroy in NamingContext 124
destroy_on_unregister in OAD 117
disconnect_pull_supplier 138
disconnect_push_consumer in PullConsumer 137
disconnect_push_supplier in PushSupplier 139
effective_profile in ClientRequestInfo 215
effective_target in ClientRequestInfo 215
encode in Codec 219
encode_value in Codec 219
establish_components in IORInterceptor 226
exceptions in RequestInfo 233
extraction methods in Any 55
for_suppliers in EventChannel 134
forward_reference in RequestInfo 233
full_poa_name in IORInfoExt 226
get_cluster_manager in NamingContextFactory 131
get_effective_component in ClientRequestInfo 215
get_effective_components in ClientRequestInfo 215
get_effective_policy in IORInfo 224
get_implementation in OAD 117
get_reply_service_context in RequestInfo 233
get_request_policy in ClientRequestInfo 215
get_request_service_context in RequestInfo 233
get_server_policy in ServerRequestInfo 236
get_slot in Current 221
getslot in RequestInfo 233
get_status in OAD 117
get_status_all in OAD 117
get_status_interface in OAD 117
IDLType 95
insertion methods in Any 56
InterfaceDef 97
IObject 99
list_all_roots in NamingContextFactory 131
lookup_by_name in EventChannelFactory 135
lookup_id in Repository 105
manager_id in IORInfo 224
name in Interceptor 223
NamedValue 65
new_context in NamingContext 124
NVList 66
object_id in ServerRequestInfo 236
obtain_pull_consumer in SupplierAdmin 139
obtain_pull_supplier 133
obtain_push_consumer in SupplierAdmin 139
obtain_push_supplier 133
operation in RequestInfo 233
operation_context in RequestInfo 233
OperationDef 101
ORB 24
orb_id in ORBInitInfo 229

- orb_id in ServerRequestInfo 236
- POA 30
- POAManager 40
- post_init in ORBInitializer 228
- pre_init in ORBInitializer 228
- Principal 41
- pull in PullSupplier 138
- rebind in NamingContext 124
- rebind_context in NamingContext 124
- received_exception in ClientRequestInfo 215
- received_exception_id in ClientRequestInfo 215
- receive_exception in ClientRequestInterceptor 217
- receive_other in ClientRequestInterceptor 217
- receive_reply in ClientRequestInterceptor 217
- receive_request in ServerRequestInterceptor 238
- receive_request_service_contexts in
 - ServerRequestInterceptor 238
- reg_implementation in OAD 117
- register_initial_reference in ORBInitInfo 229
- register_policy_factory in ORBInitInfo 229
- remove_state_contexts in
 - NamingContextFactory 131
- reply_status in RequestInfo 233
- Repository 105
- Request 69
- request_id in RequestInfo 233
- resolve in NamingContext 124
- resolve_initial_references in ORBInitInfo 229
- response_expected in RequestInfo 233
- result in RequestInfo 233
- root_context in
 - ExtendedNamingContextFactory 132
- send_exception in ServerRequestInterceptor 238
- sending_exception in ServerRequestInfo 236
- send_other in ServerRequestInterceptor 238
- send_poll in ClientRequestInterceptor 217
- send_reply in ServerRequestInterceptor 238
- send_request in ClientRequestInterceptor 217
- ServantActivator 42
- ServantBase 43
- ServantLocator 44
- server_id in ServerRequestInfo 236
- ServerRequest 72
- set_slot in Current 221
- set_slot in ServerRequestInfo 236
- state in IORInfo 224
- sync_scopoe in RequestInfo 233
- SystemException 46
- target in ClientRequestInfo 215
- target_is_a in ServerRequestInfo 236
- target_most_derived_interface in
 - ServerRequestInfo 236
- try_pull in PullSupplier 138
- unbind in NamingContext 124
- unreg_implementation in OAD 117
- unreg_interface in OAD 117
- unregister_all in OAD 117
- ModuleDef 99
 - class 99
- ModuleDescription 99
 - structure 99
- multi-threaded applications 62
- Mutex 292
 - class 292
 - methods 293

N

- NamedValue 65, 66
 - methods 65
- NamingContext
 - class 123
 - methods 124
- NamingContextExt
 - class 128
 - methods 128
- NamingContextFactory
 - class 131
 - methods 131
- Native Messaging C++
 - DuplicatedRequestTag class 211
 - PollingGroupsEmpty class 211
 - property struct 209
- native messaging C++, RequestAgent class 205
- Native Messaging for C++
 - interfaces and classes 205
 - OctetSeq class 210
 - Property fields 209
 - Property IDL definition 209
 - PropertySeq class 209
 - REPLY_NOT_AVAILABLE constant 208
 - REPLY_NOT_AVAILABLE IDL definition 209
 - ReplyRecipient class 208
 - ReplyRecipient methods 208
 - RequestAgent IDL definition 205
 - RequestAgent methods 206
 - RequestDesc fields 207
 - RequestDesc IDL definition 207
 - RequestDesc struct 207
 - RequestNotExist class 211
 - RequestTag typedef 210
 - RequestTagSeq class 210
 - typedef Cookie 210
- NativeDef class 100
- NativePriority 294
 - type 294
- Newsgroups 5
- NVLList 65, 66
 - methods 66

O

- OAD interface 116
- OAD, IDL 116
- Object 19
 - class with QoS 260
 - methods 20
 - VisiBroker extensions 22
- Object Activation Daemon, OAD interface 116
- Object Request Broker. See ORB 24
- ObjectStatus 120
- ObjectStatusList class 121
- online Help Topics, accessing 3
- OperationDef 100
 - methods 101
- OperationDescription structure 102
- OperationMode 103
 - enum 103
 - NORMAL 103
 - ONEWAY 103
- OP_NORMAL 103
- OP_ONEWAY 103

- ORB 24
 - class 24
 - extensions to CORBA 29
 - methods 24
- ORBInitializer
 - class 227
 - methods 228
- ORBInitInfo
 - BAD_INV_ORDER 229
 - class 228
 - DuplicateName 229
 - exceptions 229
 - InvalidName 229
 - members 228
 - methods 229
- overview 1

P

- Parameter struct 231
- ParameterDescription 103
 - structure 103
- ParameterList class 231
- ParameterMode 103
 - enum 103
- PDF documentation 3
- PICurrent. *See* Current
- POA
 - adapter activators 9
 - class 30
 - core classes 9
 - core interfaces 9
 - creating child POAs 9
 - methods 30
 - _POA class 8
- POALifeCycleInterceptor class 247
- POALifeCycleInterceptorManager class 248
- POAManager 38
 - methods 40
- PolicyFactory class 232
- PolicyManager class 259
- portable interceptors
 - ClientRequestInfo 214
 - POA scoped server request interceptor 225
- Portable Interceptors, interfaces 213
- PortableServer::AdapterActivator 9
- PortableServer::Current 18
- PortableServer::Current methods 19
- PortableServer_c.hh 249
- PortableServerExt_c.hh 248, 252
- PortalServerExt_c.hh 248
- PrimitiveDef 104
 - class 104
- PrimitiveKind 104
 - enum 104
- Principal 41
 - methods 41
- Priority 294
 - type 294
- PriorityMapping 295
 - class 295
 - methods 296
- PriorityModel 297
 - enum 297
- PriorityModelPolicy 297
 - class 297

- ProfileBody 269
- programming interface
 - agent 279
 - AliasDef 79
 - Any 65, 303, 315
 - ArrayDef 80
 - AttributeDef 80
 - BindInterceptor 243
 - BindOptions 10
 - BOA 11
 - ChainUntypedObjectWrapperFactory 254
 - ClientInterceptor 245
 - CompletionStatus 16
 - ConstantDef 82
 - Contained 83, 99, 100
 - Container 85, 99
 - Context 16
 - Current 291
 - DynamicImplementation 53, 307, 318
 - DynAny 53, 308, 318
 - DynArray 57, 311, 320
 - DynEnum 58, 312, 320
 - DynSequence 59, 313, 321
 - DynStruct 60, 322
 - DynUnion 61
 - EnumDef 91
 - Environment 62
 - Exception 19, 45
 - ExceptionDef 92
 - Fail 285
 - IDLType 95, 99
 - InterfaceDef 96
 - IObject 99
 - MarshalInBuffer 275
 - MarshalOutBuffer 271, 275
 - ModuleDef 99
 - ModuleDescription 99
 - Mutex 292
 - NamedValue 65
 - NativePriority 294
 - NVList 65, 66
 - Object 19
 - OperationDef 100
 - ORB 24
 - PrimitiveDef 104
 - Principal 41
 - Priority 294
 - PriorityMapping 295
 - PriorityModel 297
 - PriorityModelPolicy 297
 - Repository 105
 - Request 69, 72, 75
 - RTORB 298
 - Seq 287
 - SeqSeq 288
 - SequenceDef 106
 - SThreadPoolId 301
 - SThreadPoolPolicy 301
 - StringDef 107
 - StructDef 107
 - SystemException 45
 - TriggerHandler 286
 - TypedDef 108
 - UnionDef 109
 - UntypedObjectWrapper 256
 - UntypedObjectWrapperFactory 257

- VISInit 289
- WstringDef 114
- ProxyPullConsumer
 - interface 136
- ProxyPullConsumer interface 136
- ProxyPullSupplier interface 136
- ProxyPushConsumer interface 136
- ProxyPushSupplier interface 137
- PRTORB 298
- PullConsumer interface 137
- PullSupplier
 - interface 138, 139
 - methods 138
- PushConsumer interface 138
- PushSupplier interface 139

Q

- Quality of Service, QoS 259
- querying an object's state 19

R

- Real-Time CORBA classes 291
- RebindPolicy class 262
- RefCountServantBase methods 42
- ReplyHeader 267
- reporting
 - standard system errors 45
 - system exceptions 62
 - user exceptions 62
- Repository 105
 - methods 105
- Request 69, 72, 75
 - methods 69
- request interceptors
 - client 216
 - server 238
- RequestHeader 268
- RequestInfo
 - class 232
 - methods 233
- returning an object's Typecode 95
- RTORB
 - class 298
 - methods 299

S

- Seq methods 287
- SeqSeq methods 288
- SequenceDef 106
 - class 106
- ServantActivator
 - class 42
 - methods 42
- ServantBase methods 43
- ServantLocator
 - class 44
 - methods 44
- ServantManager class 45
- Server Manager
 - container interface 141
 - container methods for C++ 141
 - storage interface 144

- ServerRequest
 - methods 72
- ServerRequest methods 72
- ServerRequestInfo
 - BAD_INV_ORDER 235
 - class 235
 - exceptions 235
 - methods 236
- ServerRequestInterceptor
 - class 238, 249
 - methods 238
- ServerRequestInterceptorManager class 251
- setting an object's state 19
- skeletons 8
- Software updates 5
- StringDef 107
 - class 107, 109
- struct
 - Codec encoding 221
 - Parameter 231
 - UnionMember 110
- struct BindOptions 10
- struct Parameter 231
- StructDef 107
 - class 107
- StructMember structure 108
- structure
 - AttributeDescription 81
 - BindOptions 10
 - Desc 284
 - ExceptionDescription 92
 - FullInterfaceDescription 93
 - FullValueDescription 94
 - GIOP 265
 - InterfaceDescription 98
 - IOR 269
 - ModuleDescription 99
 - OperationDescription 102
 - ParameterDescription 103
 - StructMember 108
 - TriggerDesc 285
 - TypeDescription 108
 - UnionMember 110
 - ValueDescription 113
 - VersionSpec 110
- structure ExceptionDescription 92
- structure ParameterDescription 103
- stubs 7
- SupplierAdmin interface 139
- Support, contacting 4
- symbols
 - brackets [] 4
 - ellipsis ... 4
 - vertical bar | 4
- system exception classes 19
- SystemException 45
 - class 45
 - defined 46
 - methods 46

T

TaggedProfile 270
TCKind 74
 descriptions 74
Technical Support, contacting 4
ThreadPoolId 301
 type 301
ThreadPoolPolicy 301
 class 301
tie class 8
TPool 15
TriggerDesc 285
TriggerHandler
 class 286
 methods 286
TSession 15
type
 NativePriority 294
 Priority 294
 ThreadPoolId 301
TypeCode
 constructors 75
 methods 75
TypedefDef 108
 class 108
TypeDescription 108
TypeMismatch class 219

U

UnionDef 109
UnionMember 110
 structure 110

UnknownEncoding
 class 220
 CodecFactory 220
UntypedObjectWrapper 256
 class 256
UntypedObjectWrapperFactory 257
 class 257
user exception classes 19

V

ValueBoxDef class 110
ValueDef class 111
ValueDescription structure 113
_var class 8
Var classes 8
VersionSpec 110
vinit.h 289
VISClosure class 253
VISClosureData class 254
VisiBroker overview 1
VISInit 289
 methods 289, 290
vobjwrap.h 254, 256, 257

W

World Wide Web
 Borland documentation on the 5
 Borland newsgroups 5
 Borland updated software 5
WstringDef 114
 class 114

