Netscape Internet Service Broker for C++ Reference Guide

Version 1.0


Recycled and Recyclable Paper

Version 1.0

Printed in USA
99 98 97 10 9 8 7 6 5 4 3 2 1

Netscape Communications Corporation 501 East Middlefield Road, Mountain View, CA 94043

Contents

Organization of this Guide vii
Typographic Conventions vii
Platform Conventions viii
Where to Find Additional Information viii

Chapter 1 Commands 1
General Information 1
idl2ir 2
Description 2
irep 3
Description 3
listimpl 3
Description 4
oad 5
Description 5
orbeline 6
Description 6
regobj 8
Description 8
Server Activation Policies 9
unregobj 9
Description 9
Windows Implementation of Commands 10
listimpl 10
regobj 12
unregobj 15

Chapter 2 Programming Interfaces 17
ActivationImplDef 18
Include File 19

IDL Definition 19
Methods 19
Any 21
Include File 21
BOA 21
Include File 22
Methods 22
ClientEventHandler 28
Include File 28
Methods 29
Contained 30
Include File 31
IDL Definition 31
Methods 31
Container 33
Include File 34
IDL Definition 34
Methods 35
Context 40
Include File 40
Methods 40
CreationImplDef 44
Include File 44
IDL Definition 44
Methods 45
Environment 47
Include File 48
Methods 48
Exception 50
Include File 50
HandlerRegistry 50
Include File 51
Methods 51

IDLType 53
Include File 53
IDL Definition 53
Methods 54
ImplementationDef 54
Include File 54
IDL Definition 54
Methods 55
ImplEventHandler 57
Include File 57
Methods 58
InterfaceDef 60
Include File 60
IDL Definition 60
Methods 61
IRObject 63
Include File 63
IDL Definition 64
Methods 64
NamedValue 64
Include File 64
Methods 65
NVList 65
Include File 66
Methods 66
Object 68
Include File 69
Methods 69
OperationDef 74
Include File 74
IDL Definition 74
Methods 75

ORB 77
Include File 77
Methods 77
Principal 85
Include File 85
Methods 85
Repository 86
Include File 86
IDL Definition 86
Methods 86
Request 88
Include File 88
Methods 88
SystemException 90
Include File 90
Methods 90

Preface

The *Netscape Internet Service Broker for C++ Reference Guide* provides reference information about commands and interfaces in Netscape Internet Service Broker for C++ (ISB for C++). This Preface lists the contents of the *Netscape Internet Service Broker for C++ Reference Guide*, describes typographic and syntax conventions used throughout the guide, and provides references for more information about ISB for C++ and CORBA.

Organization of this Guide

Typographic Conventions

Platform Conventions

Where to Find Additional Information

Organization of this Guide

This guide includes the following sections:

- "Commands" provides detailed information about commands in ISB for C++.
- "Programming Interfaces" provides information about the programming interfaces in ISB for C++. It includes the classes and structures that can be used to create a client application or object implementation.

Typographic Conventions

This guide uses the following conventions:

Convention	Used for
boldface	In syntax diagrams, bold type indicates a required item.
italics	Italics indicates information that the user or application provides, such as variables in syntax diagrams. It is also used to introduce new terms.
`monospace`	Monospace typeface is used for sample command lines, code, and object names.
[ ]	Brackets indicate optional items in syntax diagrams
...	An ellipsis indicates that the previous argument can be repeated.
|	A vertical bar separates two mutually exclusive choices.
. . .	A column of three dots indicates the continuation of previous lines of code.

Platform Conventions

This guide uses the following symbols to indicate that information is platform-specific:

W All Windows platforms including Windows 3.1, Windows NT, and Windows 95.

NT Windows NT only.

95 Windows 95 only.

U All Unix platforms.

Where to Find Additional Information

For more information about Netscape Internet Service Broker for C++, refer to the following information sources:

- ***Netscape Internet Service Broker for C++ Programmer's Guide***. This guide provides information on developing distributed object-based applications in C++ for Windows and UNIX platforms.

For more information about the CORBA specification, refer to the following sources:

- ***The Common Object Request Broker: Architecture and Specification*** - 96-03-04. This document is available from the Object Management Group and describes the architectural details of CORBA. You can access the CORBA specification using the World Wide Web at the following URL: http://www.omg.org/corba/corbiiop.htm.
- ***IDL to C++ Language Mapping*** - 94-9-14. This document is available from the Object Management Group and describes the Interface Definition Language mappings for C++.

Chapter

1

Commands

This chapter covers commands used for creating, executing, and managing applications developed with Netscape Internet Service Broker for C++ (ISB for C++). It includes the following major sections:

- General Information
- idl2ir
- irep
- listimpl
- oad
- orbeline
- regobj
- unregobj
- Windows Implementation of Commands

General Information

For Windows users, to view options for a command, enter:

`command name -?`

For example: `orbeline -?`

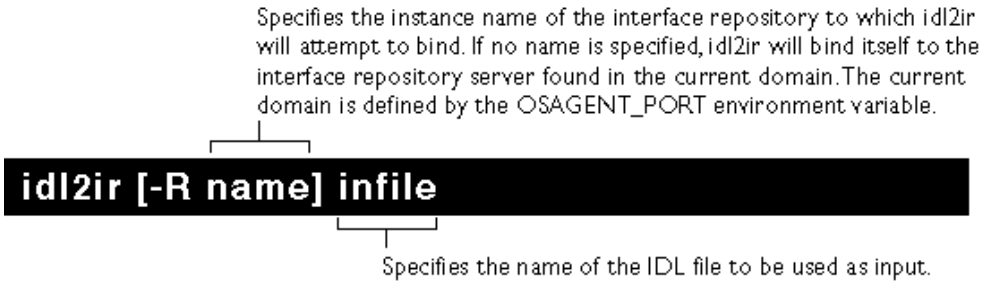
For UNIX users, to view options for a command, enter:

`command name -\?`

For example: `orbeline -\?`

idl2ir

This command allows you to populate an interface repository with objects defined in an Interface Definition Language source file.



Description

The **idl2ir** command takes an IDL file as input, binds itself to an interface repository server, and populates the repository with the IDL constructs contained in infile. If the repository already contains an item with the same name as an item in the IDL file, the old item is replaced.

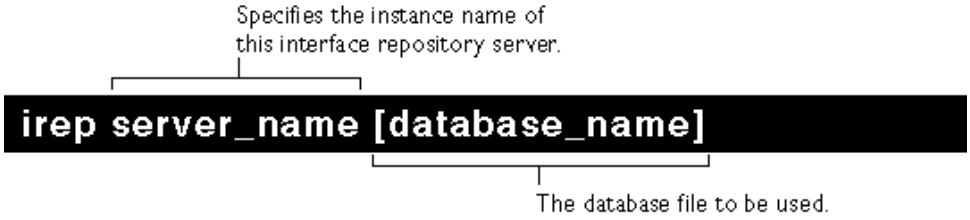
Note The `idl2ir` command does not handle anonymous arrays or sequences properly. To work around this problem, `typedefs` must be used for all sequences and arrays.

Example:

```
idl2ir -R my_repository library/lib.idl
```

irep

This command starts an interface repository server.



Description

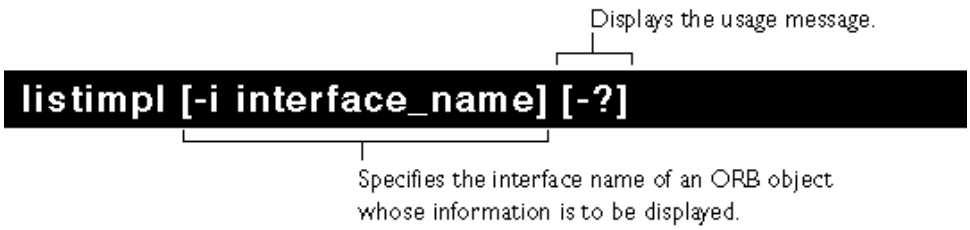
The **irep** command starts an interface repository server that manages a database containing detailed descriptions of IDL interfaces. The repository's database includes interface names, inheritance structure, supported operations, and arguments. The interface repository can consist of multiple databases and can be loaded using the **idl2ir** command. Applications can bind to and query the interface repository using the Repository class.

Example:

```
irep my_server
```

listimpl

This command allows you to list all ORB object implementations registered with the Object Activation Daemon.



Description

This command lists information in the OAD's implementation repository. The information for each object includes:

- Interface names of the ORB objects.
- Instance names of the object or objects offered by that implementation.
- Full pathname of the server implementation's executable.
- Activation policy of the ORB object (shared, unshared, or per-method).
- Reference data specified when the implementation was registered with the OAD.
- List of arguments to be passed to the server at activation time.
- List of environment variables to be passed to the server at activation time.

For UNIX, if interface_name is specified, only information for that ORB object is displayed; otherwise all ORB objects registered with the OAD and their information is displayed.

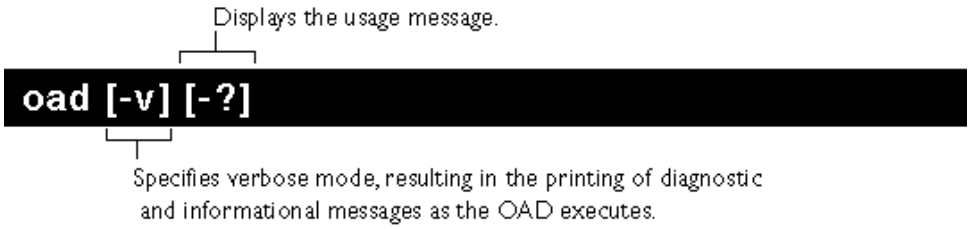
The implementation repository files are assumed to reside in the **impl_dir** subdirectory whose path is defined by the **ORBELINE** environment variable. A different directory name can be set using the **ORBELINE_IMPL_NAME** environment variable. The path to this directory can be changed using the **ORBELINE_IMPL_PATH** environment variable.

Example:

```
listimpl -i Library
```

oad

This command starts the Object Activation Daemon



Description

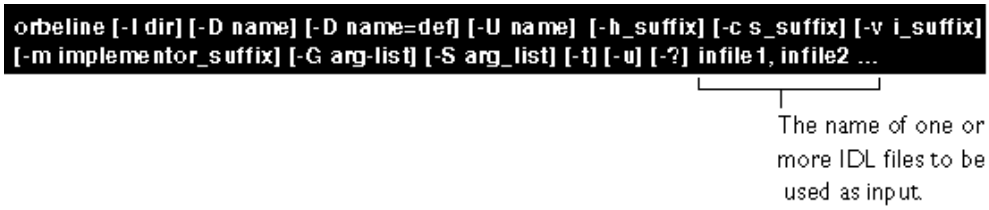
Each object that is to be activated automatically must register its implementation with the OAD. When an object is registered with the OAD, ISB for C++ is notified that the object exists. When a client application binds to such an object, ISB for C++ locates the object and returns the OAD with which the object was registered. ISB for C++ then negotiates with the OAD to activate the requested object. This activation process is transparent to the application program.

Example:

```
oad -v
```

orbeline

This command implements ORBeline's IDL to C++ compiler.



Description

This command takes IDL files as input and generates the corresponding C++ classes for the client and server side, as well as client stubs and server skeleton code.

Example:

```
orbeline -h hx -m _serv -S tie -S excepspec lib.idl
```

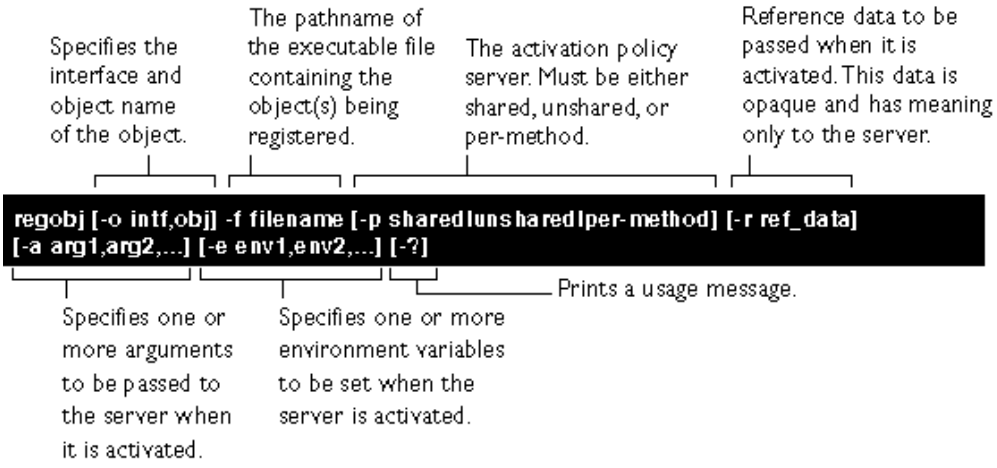
Argument	Description
-o *dir*	Directs output to the directory specified by *dir*.
-I *dir*	Adds the specified directory name to the beginning of the search path for include files that do not begin with a "/" character.
-D *name*	Defines the specified name just as though it had been defined with a `#define` directive specifying a value of 1.
-D *name=def*	Defines the specified name just as though it had been defined using the #define directive.
-U *name*	Removes the initial definition of the specified `name`, where `name` is a symbol that is predefined.
-h *header_suffix*	Sets the suffix for include files generated by the IDL compiler. The suffix ".hh" is used by default.

Argument	Description
-c *source_suffix*	Sets the suffix for source files generated by the IDL compiler. The suffix ".cc" is used by default.
-m *implementor_suffix*	Sets the suffix for server-side files generated by the IDL compiler. The suffix "_server" is used by default for Unix platforms. The suffix "_s" is used by default for Windows platforms.
-v *invoker_suffix*	Sets the suffix for client-side files generated by the IDL compiler. The suffix "_client" is used by default for Unix platforms. The suffix "_c" is used by default on Windows platforms.
-G stdstream	Generate operators for writing objects to and reading objects from a stream. These operators can be used to write the objects to stdout or a file. This option is ON by default.
-G container	Generate methods for placing ORB objects into containers. These methods include compare, hash, and assignment operators. This option is ON by default.
-G excepspec	Generate exception specification code. For each method defined, the list of exceptions which it can throw are specified as part of the method declaration This option is ON by default.
-G exceptions	Generate code that uses native C++ language exceptions. This option is ON by default.
-G tie	Generate code to enable the tie or delegation mechanism to be used. This option is ON by default.
-S stdstream	Suppress the generation of class stream operators.
-S container	Suppress the generation of code that places ORB objects into containers.
-S excepspec	Suppress the generation of exception specification code.
-S exceptions	Suppress the generation of code that uses native C++ language exceptions.
-S tie	Suppress the generation of code that enables the tie or delegation mechanism.
-t	Generate typecodes for interfaces and types defined in IDL.

Argument	Description
-u	Prints a usage message.
-?	Prints a usage message.

regobj

This command allows you to register one or more ORB object implementations with the Object Activation Daemon. For more information on the OAD, see page 1-7 in this guide.



Description

If the OAD is running, this command registers one or more ORB objects with the Object Activation Daemon. Once registered, these objects can be activated automatically by the OAD when a client requests to bind to the object. Also, ISB for C++ is notified that the objects are available through the OAD. The object's implementation is also added to ISB for C++'s implementation repository.

Note If an object implementation is started manually as a persistent server, it does not need to be registered with the OAD.

Example:

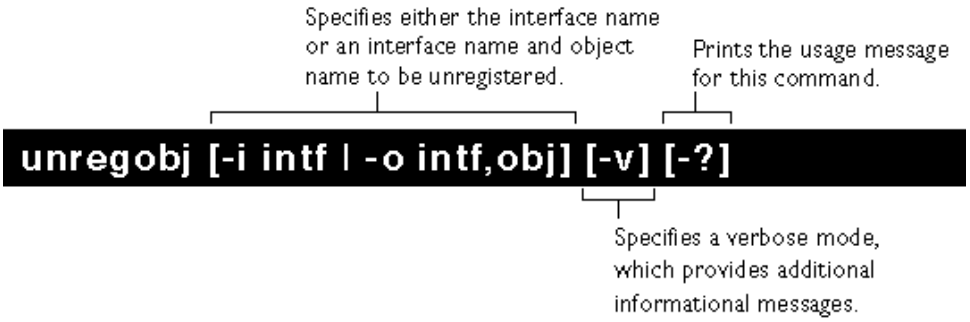
```
regobj -o Library,Harvard -f lib_server -p shared
```

Server Activation Policies

shared	Multiple objects of a given implementation share the same server.
unshared	Only one object of a given implementation can be active in a server at one time. If multiple clients wish to bind to the same object implementation, a separate server is activated for each client application. A server exits when its client application disconnects or exits.
per-method	Each invocation of a method results in a new server being activated. The server exits when the method invocation completes.

unregobj

This command unregisters ORB objects registered with the Object Activation Daemon. For more information on the OAD, see page 1-7 in this guide.



Description

This command unregisters one or more ORB objects with the Object Activation Daemon. Once an object is unregistered, it can no longer be activated automatically by the OAD when a client requests the object. ORB objects being unregistered must have been previously registered using the **regobj** command, described on page 1-12 in this guide.

If you specify only an interface name, all ORB objects with that interface that are registered with the OAD will be unregistered. Alternatively, you can specifically identify an ORB object by its interface name and object name.

Note If an object implementation is started manually as a persistent server, it does not need to be registered with the OAD.

Example:

```
unregobj -o Library,Harvard
```

Example:

```
unregobj -i Library
```

Windows Implementation of Commands

This section shows the Windows implementation of the following commands.

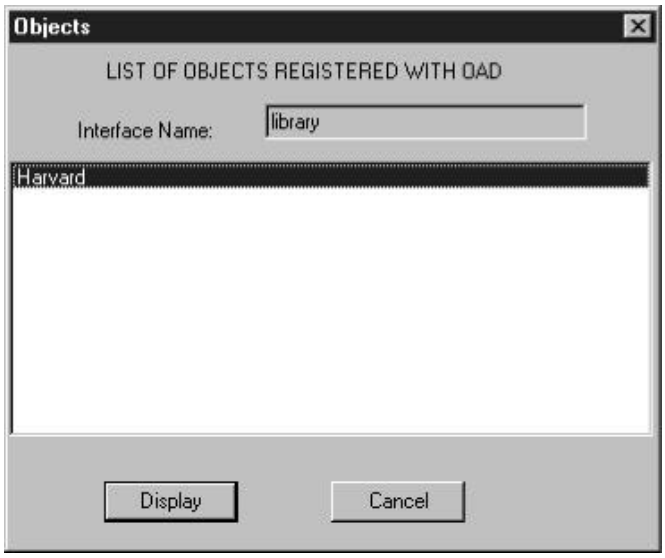
- listimpl
- regobj
- unregobj

For more detailed information about a command, see the corresponding section earlier in this chapter.

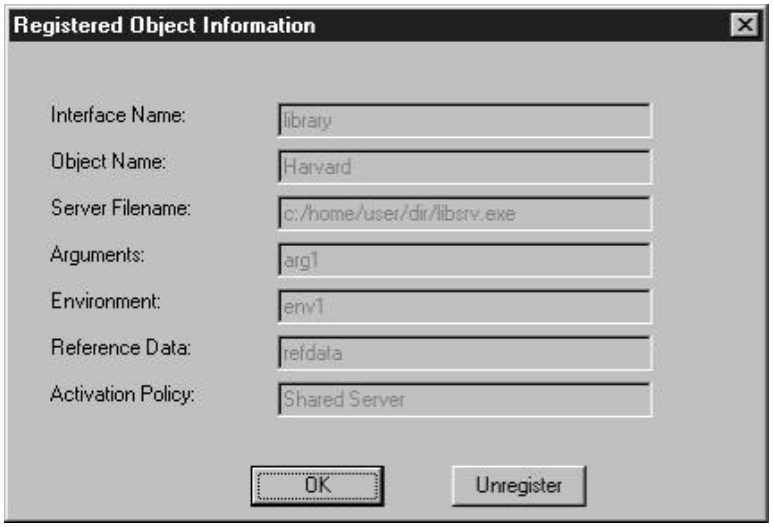
listimpl

In Windows 95 or Windows NT, you list registered objects by launching the ORBeline Registry icon first and then selecting List in the menu bar. The following screen appears:

This window lists the interfaces that are registered with the object activation daemon on your computer. By selecting one of the interfaces and clicking on Browse, you can view the object(s) implementing that interface that are also registered with the activation daemon. The following screen appears:



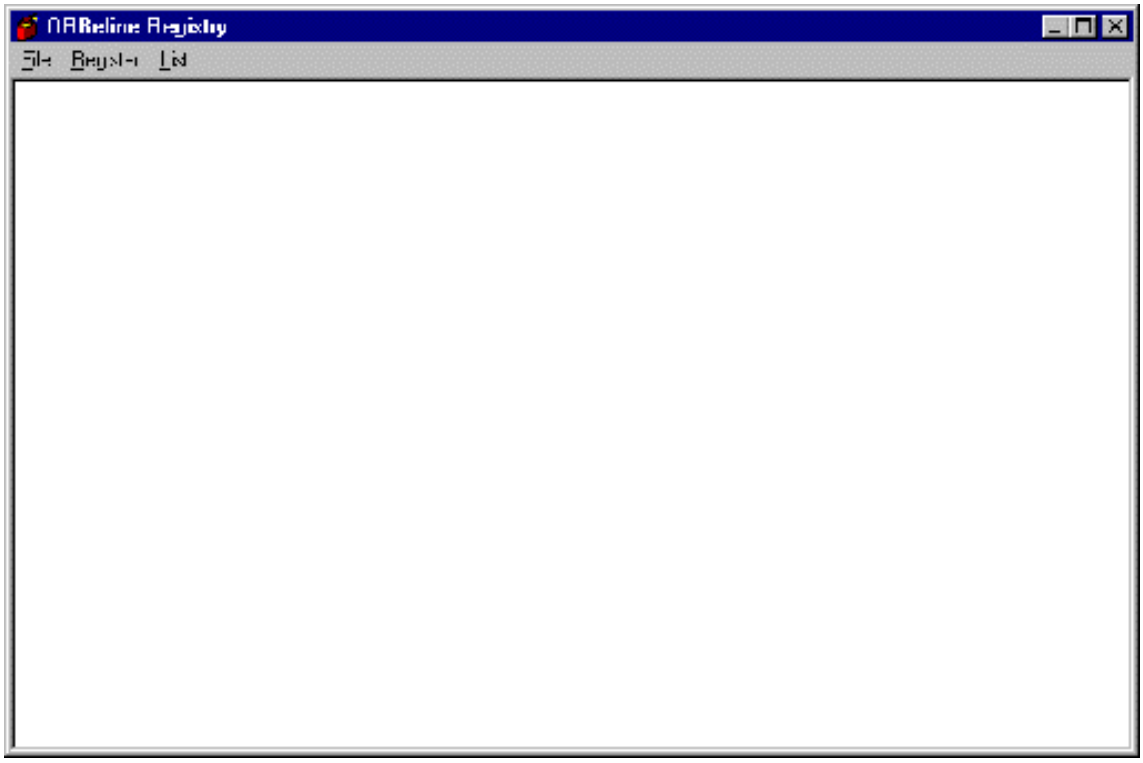
If you want to view information about this object or unregister it, click Display. The following sample of the screen appears:



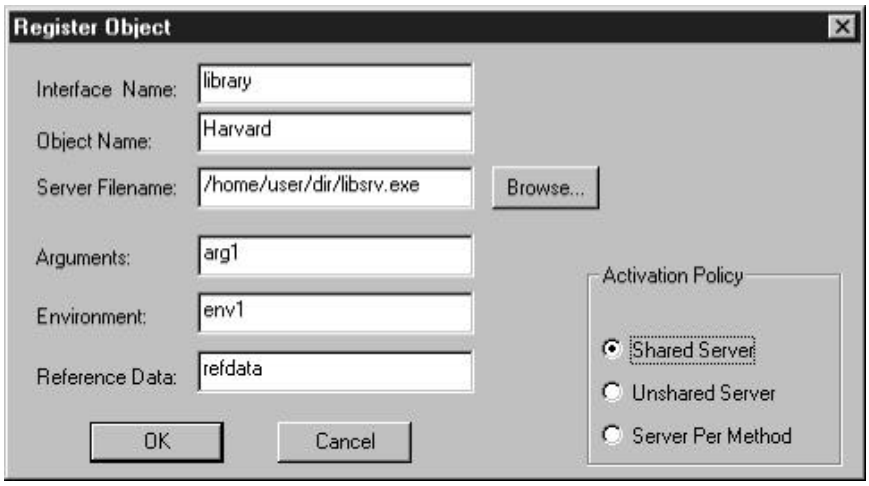
To unregister an object, click on Unregister.

regobj

In Windows 95 or Windows NT, you start the object registration command by typing **regobj** at the command line or launching the ORBeline Registry icon. The following screen appears:



You can register an object by selecting Register from the menu bar. The following sample of the screen appears:



The following table describes the fields in this dialog box.

Field	Description
Interface Name	Required. Enter the name of the object's IDL interface.
Object Name	Required. Enter the name of the object you want to register.
Server Filename	Required. Enter the name of a file to execute when the object you are registering is activated.
Activation Policy	Optional. Defaults to Shared Server. For more information about activation policies, see *Server Activation Policies* in the command section under `regobj`.
Arguments	Optional. Enter the arguments, if any, that should be passed to the executable at startup time. These arguments are separated by commas.
Environment	Optional. Enter a list of environment variables, separated by commas, and their entries. For example: ORBELINE=c:orbeline,LIB=c:orblib.
Reference Data	Optional. Enter any application specific data relevant to this application. At runtime, each object can query the Basic Object Adaptor (BOA) for reference data.

unregobj

In Windows 95 or Windows NT, you unregister objects by displaying the information about an object. To learn how to display information about an object, see the Windows information for listimpl.

Chapter

2

Programming Interfaces

This chapter covers the programming interfaces provided by ISB for C++. It includes the classes and structures you can use to create a client application or object implementation.

- ActivationImplDef
- Any
- BOA
- ClientEventHandler
- Contained
- Container
- Context
- CreationImplDef
- Environment
- Exception
- HandlerRegistry
- IDLType

- ImplementationDef
- ImplEventHandler
- InterfaceDef
- IRObject
- NamedValue
- NVList
- Object
- OperationDef
- ORB
- Principal
- Repository
- Request
- SystemException

ActivationImplDef

```
class ActivationImplDef : public ImplementationDef
```

The `ActivationImplDef` class is used by object implementations that defer the instantiation of an object until the first client request arrives. This class embeds an interface and object name for a particular ORB object as well an object of class `Activator`. When an object of this class is passed to the `BOA::obj_is_ready` method, the `activate` and `deactivate` methods used by the `BOA` are overridden by the `Activator` object's methods. To use `ActivationImplDef`, you must provide an `Activator` class along with the implementation for its methods.

See also the BOA class and the `ImplementationDef` class.

Include File

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

IDL Definition

```
interface ActivationImplDef: ImplementationDef
{
    Attribute Activator activator_obj;
};
```

Methods

- `ActivationImplDef`(const char *`interface_name`, const char *`object_name`, const ReferenceData& `id`, Activator_ptr `act`)
- ~ActivationImpDef
- lactivator_obj
- _duplicate
- _narrow
- _nil

```
ActivationImplDef(const char *interface_name,
                  const char *object_name,
                  const ReferenceData& id,
                  Activator_ptr act)
```

The constructor is used by an object implementation to create an `ActivationImplDef` for a particular C++ class that will implement an ORB object.

Parameter	Description
interface_name	The object's interface name.
object_name	The object's instance name.
id	Reference data can be used to distinguish between several instances of objects that have the same name. It is chosen by the object implementation and remains constant throughout the object's lifetime.

```
~ActivationImplDef()
```

Destructor for this object.

```
Activator_ptr activator_obj();
```

This method returns a pointer to the `Activator` object associated with this object.

```
void Activator_ptr activator_obj(Activator_ptr val);
```

This method sets the `Activator` object to be used by this object.

Parameter	Description
val	A pointer to an Activator object to be used by this object.

```
static ActivationImplDef_ptr _duplicate(ActivationImplDef_ptr obj);
```

Duplicates the `ActivationImplDef` object pointed to by the parameter.

Parameter	Description
obj	Pointer to the object to be duplicated.

```
static ActivationImplDef_ptr _narrow(ImplementationDef_ptr ptr);
```

This method attempts to narrow the `ImplementationDef_ptr` that is passed in to an `ActivationImplDef_ptr`. If the supplied pointer does not point to an object of `ActivationImplDef`, a NULL value is returned. Otherwise, an `ActivationImplDef` pointer is returned.

Parameter	Description
ptr	A pointer to `ImplementationDef` object.

```
static ActivationImplDef_ptr _nil();
```

This method returns a NULL pointer cast to an ActivationImplDef_ptr.

Any

```
class Any
```

This class is used to represent any IDL type so that any value can 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 provide to construct, copy, and destroy an object as well as initialize and query the object's type and value. In addition, streaming operators are provided to read and write the object to a stream.

Include File

Include `any.h` when you use this structure.

BOA

```
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.

Include File

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

Methods

- BOA_init
- change_implementation
- create
- deactivate_impl
- deactivate_obj
- dispose
- _duplicate
- get_id
- get_prinicpal
- impl_is_ready
- _nil
- obj_is_ready
- release
- scope

```
static BOA_ptr ORB::BOA_init(int& argc,
                             char *const *argv,
                             const char *boa_identifier = "PMC_BOA")
```

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

Parameter	Description
argc	The number of arguments passed.
argv	An array of char pointers to the arguments. All but two of the arguments take the form of a keyword and a value and are shown below. This method will ignore any keywords that it does not recognize.

-OAipaddr *ip_address*	Specifies the IP address to be used for this BOA. Useful if the host has more than one network interface. If this option is not specified, the host's default IP address is used.
-OAport port_num	Specifies the port number to use for this BOA. If not specified, an unused port number is used.
-OAsendbufsize *size*	Specifies the size in bytes of the network transport's send buffer. If this option is not specified, an appropriate buffer size is used.
-OArcvbufsize *size*	Specifies the size in bytes of the network transport's receive buffer. If this option is not specified, an appropriate buffer size is used.
-OAnoshm	Disables the use of shared memory for transmitting messages when the client and object implementation processes are located on the same host.
-OAshm	Enables the use of shared memory for transmitting messages.

```
void change_implementation(Object_ptr obj, ImplementationDef_ptr impl)
```

This method changes the implementation definition associated with the specified object. You should use this method with caution. The implementation name should not be changed and you must ensure that the new

implementation definition specifies the same type of object as the original definition. If the `ImplementationDef_ptr` does not point to a `CreationImplDef` pointer, this method will fail.

Parameter	Description
obj	A pointer to the object whose implementation is to be changed.
impl	A pointer to the new implementation definition for this object. This must actually be a `CreationImplDef_ptr` cast to an `ImplementationDef_ptr`.

```
Object_ptr create(const ReferenceData& ref_id,
                  InterfaceDef_ptr intf_def,
                  ImplementationDef_ptr impl_def)
```

This method creates an ORB object and returns a reference to the newly created object. Once the object reference is created, the client application can invoke methods on object reference.

Parameter	Description
ref_id	This parameter is not used, but is provided for compliance with the CORBA specification.
intf_def	This parameter is not used, but is provided for compliance with the CORBA specification.
impl_def	This pointer's true type is `CreationImplDef` and provides the interface name, object name, path name of the executable and activation policy, along with other parameters.

```
void deactivate_impl(ImplementationDef_ptr impl_def)
```

This method deactivates the implementation specified by the ImplementationDef_ptr. After this method is called, no client requests will be delivered to the object within this implementation until the objects and implementation are activated using the `obj_is_ready` and `impl_is_ready` methods.

Parameter	Description
impl_def	This pointer's true type is CreationImplDef and provides the interface name, object name, path name of the executable and activation policy, along with other parameters. See X for a complete discussion of the CreationImplDef class.

```
void deactivate_obj(Object_ptr obj)
```

This method notifies the BOA that the specified object is to be deactivated. After this method is invoked, the BOA will not deliver any requests to the object until `obj_is_ready` or `impl_is_ready` is invoked.

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

```
void dispose(Object_ptr obj)
```

This method unregisters the implementation of the specified object from the Object Activation Daemon. After this method is invoked, all references to this object will be invalid and any connections to this object implementation will be broken. If the object is allocated, the application must delete the object.

Parameter	Description
obj	Pointer to the object to be unregistered.

```
static BOA_ptr _duplicate(BOA_ptr ptr)
```

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

Parameter	Description
ptr	A BOA pointer.

```
ReferenceData get_id(Object_ptr obj)
```

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. Reference data is often used to distinguish between object implementations that have the same name.

Parameter	Description
obj	A pointer to the object whose reference data is to be returned.

```
Principal_ptr get_principal(Object_ptr obj, Environment_ptr env)
```

This method returns the Principal object associated with the specified object.

Parameter	Description
obj	A pointer to the object whose implementation is to be changed.
env	A pointer to the Environment object associated with this Principal.

```
void impl_is_ready(ImplementationDef_ptr impl_def=NULL)
```

This method notifies the BOA that one or more objects in the server are ready to receive service requests.If all objects that the implementation is offering have been created through C++ instantiation and activated using the `obj_is_ready` method, the `ImplementationDef_ptr` should not be specified.

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 for its single object.

Parameter	Description
impl_def	This pointer's true type is `ActivationImplDef(const char *interface_name, const char *object_name, const ReferenceData& id, Activator_ptr act)` and provides the interface name, object name, path name of the executable and activation policy, along with other parameters.

```
void obj_is_ready(CORBA::Object_ptr obj,
                  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 an pointer to an `ActivationImplDef` object that has been initialized. For more information, see `ActivationImplDef`(const char *`interface_name`, const char *`object_name`, const ReferenceData& `id`, Activator_ptr `act`).

Parameter	Description
obj	A pointer to the object to be activated.
impl_ptr	A optional pointer to an ActivationImplDef object.

```
static BOA_ptr _nil()
```

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

```
static void CORBA::release(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
boa	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 local or global. 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
val	The new registration scope for the BOA, either SCOPE_LOCAL or SCOPE_GLOBAL.

ClientEventHandler

```
class ClientEventHandler
```

The `ClientEventHandler` class defines the interface for registering an event handler for client applications. Event handlers provide a set of methods that are invoked by the ORB when one of the following events occurs.

- A bind to an object succeeds.
- A bind fails.
- A server aborts.
- A rebind succeeded, following a server abort.
- A rebind failed.

You derive your own event handling class from `ClientEventHandler` and provide the implementation just for the event handling methods in which you are interested. Your implementation will be invoked by the ORB when a particular event occurs and can be used to implement logging or security features that you design. If you are not interested in handling a particular event, do not declare its method in your derived class.

After creating an event handler, your client application must register it using the `HandlerRegistry` class. You can register an event handler for a particular object or a global event handler for all objects.

If both global and object-specific event handlers are registered and an event occurs on the object with its own event handler, the object-specific handler will take precedence over the global handler.

See ImplEventHandler in this guide for information on creating event handlers for servers.

Include File

Include the `corba.h` and `pmcext.h` files when you use this class.

Methods

- bind_failed
- bind_succeeded
- rebind_failed
- rebind_succeeded
- server_aborted

```
virtual void bind_failed(CORBA::Object_ptr)
```

You can choose to implement this method, which will be invoked by the ORB when a bind fails. If the event handler was registered for a particular object, it will only be invoked when a bind fails for that object. If the event handler is registered with a global scope, it will be called when any bind issued by the client fails.

Parameter	Description
CORBA::Object_ptr	A reference to the object that could not be bound.

```
virtual void bind_succeeded(CORBA::Object_ptr, const ConnectionInfo&)
```

You can choose to implement this method, which will be invoked by the ORB when a bind succeeds. If the event handler was registered for a particular object, it will only be invoked when a bind succeeds for that object. If the event handler is registered with a global scope, it will be called when any bind issued by the client succeeds.

Parameter	Description
CORBA::Object_ptr	A reference to the object that has been successfully bound.
ConnectionInfo	This structure contains the host name, port, and file descriptor. You can use this information to adjust the characteristics of the connection.

```
virtual void rebind_failed(CORBA::Object_ptr)
```

You can choose to implement this method, which will be invoked by the ORB when a rebind fails. If the event handler was registered for a particular object, it will only be invoked when a rebind fails for that object. If the event handler is registered with a global scope, it will be called when any rebind issued on behalf of the client fails. The rebind option must be enabled.

Parameter	Description
CORBA::Object_ptr	A reference to the object that could not be rebound.

```
virtual void rebind_succeeded(CORBA::Object_ptr, const ConnectionInfo&)
```

You can choose to implement this method, which will be invoked by the ORB when a rebind succeeds. If the event handler was registered for a particular object, it will only be invoked when a rebind succeeds for that object. If the event handler is registered with a global scope, it will be called when any bind issued by the client succeeds. The rebind option must be enabled.

Parameter	Description
CORBA::Object_ptr	A reference to the object that has been successfully rebound.
ConnectionInfo	This structure contains the host name, port, and file descriptor. You can use this information to adjust the characteristics of the connection.

```
virtual void sever_aborted(CORBA::Object_ptr)
```

You can choose to implement this method, which will be invoked by the ORB whenever a connection to a server is lost. If the event handler was registered for a particular object, it will only be invoked when the connection is lost for that object. If the event handler is registered with a global scope, it will be called when any of the client's connections are lost.

Parameter	Description
CORBA::Object_ptr	A reference to the object whose connection was lost.

Contained

```
class Contained : public IRObject
```

The `Contained` class is used to derive all Interface Repository (IR) objects that are themselves contained within another IR 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 `corba.h` when you use this class.

IDL Definition

```
interface Contained: IRObject {
  attribute RepositoryId id;
  attribute Identifier name;
  attribute VersionSpec 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 VersionSpec new_version
          );
};
```

Methods

- absolute_name
- containing_repository
- defined_in
- describe
- move
- name
- version

```
char * absolute_name()
```

This method returns the absolute name, which 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()
```

This method returns the `Repository` that is eventually reached by recursively following the `defined_in` attribute.

```
CORBA::Container_ptr defined_in()
```

This method returns the `Container` object in which this object resides.

```
CORBA::Contained::Description *describe()
```

This method returns a structure that describes the interface, such as the type returned and its associated value.

```
void move(CORBA_Container_ptr new_CORBA_container,
          const char * new_name,
          const CORBA_VersionSpec& version)
```

This method removes this object from its current container and adds it to the specified container. The new `Container` must be in the same repository as the original and capable of containing the object's type. Further, the new `Container` must not already contain an object with the same name.The object's `defined_in` and `absolute_name` attributes are updated to reflect the new container. If this object is itself a `Container`, all of it's contained object's are also updated.

Parameter	**Description**
new_CORBA_container	The Container where the object is to be moved.
new_name	The object's name.
version	The container's version.

```
char *name()
```

This method returns the name of the object. The name uniquely identifies it 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::VersionSpec version()
```

This method returns the object's version. The version distinguishes this object from other objects that have the same name.

```
void version(CORBA::VersionSpec& val)
```

This method sets this object's version.

Parameter	**Description**
val	The object's id.

Container

```
class Container : public IRObject
```

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 all types of IDL types defined in `orbtypes.h`, including `InterfaceDef`, `ModuleDef` and `ConstantDef` classes. Each definition that is created will have its `defined_in` attribute initialized to point to this object.

Include File

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

IDL Definition

```
interface Container: IRObject {
  ContainedSeq contents(
          in DefinitionKind limit_type,
          in boolean exclude_inherited
  );
  Contained lookup(in ScopedName search_name);
  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
);
```

Methods

- contents
- create_alias
- create_constant
- create_enum
- create_exception
- create_interface
- create_module
- create_struct
- create_union
- describe_contents
- lookup
- lookup_name

```
ContainedSeq * contents(DefinitionKind limit_type,
                        Boolean exclude_inherited)
```

This method returns a list of contained object definitions directly contained or inherited into the container. You can use this method to navigate through the hierarchy of object definitions in the `Repository`. All object definitions contained by modules in the `Repository` are returned, followed by all object definitions contained within each of those modules.

Parameter	**Description**
limit_type	The interface object types to be returned. Specifying `dk_all` will return objects of all types.
exclude_inherited	If set to true, inherited objects will not be returned.

```
AliasDef_ptr create_alias(const char* id,
                          const char* name,
                          const CORBA_VersionSpec& version,
                          IDLType_ptr original_type)
```

This method creates an `AliasDef` object in this `Container` with the specified attributes and returns a pointer to the newly created object.

Parameter	Description
id	The alias id.
name	The alias name.
version	The alias version.
original_type	The type of the object for which this object is an alias.

```
ConstantDef_ptr create_constant(const char * id,
                                const char * name,
                                const CORBA_VersionSpec& version,
                                IDLType_ptr type,
                                const Any& value)
```

This method creates a `ConstantDef` object in this `Container` with the specified attributes and returns a pointer to the newly created object.

Parameter	Description
id	The constant's id.
name	The constant's name.
version	The constant's version.
type	The type of the value specified below.
value	The constant's value.

```
EnumDef_ptr create_enum(const char * id,
                        const char * name,
                        const CORBA_VersionSpec& version,
                        const EnumMemberSeq& members)
```

This method creates an `EnumDef` object in this `Container` with the specified attributes and returns a pointer to the newly created object.

Parameter	Description
id	The enumeration's id.
name	The enumeration's name.
version	The enumeration's version.
members	A list of the enumeration's fields.

```
ExceptionDef_ptr create_exception(const char * id,
                                  const char * name,
                                  const CORBA_VersionSpec& version,
                                  const EnumMemberSeq& members)
```

This method creates an `ExceptionDef` object in this `Container` with the specified attributes and returns a pointer to the newly created object.

Parameter	Description
id	The exception's id.
name	The exception's name.
version	The exception's version.
members	A list of the types of the members of the exception, if any.

```
InterfaceDef_ptr create_interface(const char * id,
                                  const char * name,
                                  const CORBA_VersionSpec& version,
                                 const InterfaceDefSeq& base_interfaces)
```

This method creates a `InterfaceDef` object in this `Container` with the specified attributes and returns a pointer to the newly created object.

Parameter	Description
id	The interface's id.
name	The interface's name.
version	The interface's version.
base_interfaces	A list of all interfaces that this interface inherits from.

```
ModuleDef_ptr create_module(const char * id,
```

```
                            const char * name,
                            const CORBA_VersionSpec& version)
```

This method creates a `ModuleDef` object in this `Container` with the specified attributes and returns a pointer to the newly created object.

Parameter	Description
id	The module's id.
name	The module's name.
version	The module's version.

```
StructDef_ptr create_struct(const char * id,
                            const char * name,
                            const CORBA_VersionSpec& version,
                            const StructMemberSeq& members)
```

This method creates a `StructureDef` object in this `Container` with the specified attributes and returns a pointer to the newly created object.

Parameter	Description
id	The structure's id.
name	The structure's name.
version	The structure's version.
members	The values for the structure's fields.

```
UnionDef_ptr create_union(const char * id,
                          const char * name,
                          const CORBA_VersionSpec& version,
                          IDLType_ptr discriminator_type,
                          const UnionMemberSeq& members)
```

This method creates a `UnionDef` object in this `Container` with the specified attributes and returns a pointer to the newly created object.

Parameter	Description
id	The union's id.

name	The union's name.
version	The union's version.
discriminator_type	The type of the union's discriminant value.
members	A list of the types of each of the union's fields.

```
DescriptionSeq * describe_contents(DefinitionKind limit_type,
                                   Boolean exclude_inherited,
                                   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 `dk_all` will return the descriptions for objects of all types.
exclude_inherited	If set to true, descriptions for inherited objects will not be returned.
max_returned_objs	The maximum number of descriptions to be returned. Setting this parameter to -1 will return all objects.

```
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.

```
ContainedSeq * lookup_name(const char *search_name,
                           Long levels_to_search,
                           DefinitionKind limit_type,
                           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 type object, 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. Setting this parameter to a value of -1 will cause all levels to be searched. Setting this parameter to 1 will search only this object.
limit_type	The interface object types to be returned. Specifying `dk_all` will return objects of all types.
exclude_inherited	If set to true, inherited objects will not be returned.

Context

```
class Context
```

The `Context` class represents 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 formal method signature, that is, 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 and chains the name/value pairs together.

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

Include File

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

Methods

- context_name
- create_child
- delete_value
- _duplicate
- get_default_context
- get_values
- is_nil
- _nil
- parent
- _release
- release
- set_one_value
- set_values

```
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, a `NULL` value is returned.

```
Status create_child(const char * name, Context_ptr& ctx)
```

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

Parameter	Description
name	The name of the new `Context` object.
ctx	A reference to the newly created child `Context`.

```
Status delete_value(const char *name)
```

This method deletes one or more properties from this object. The name may contain a trailing "*" wildcard character to delete all matching properties. A single asterisk can be specified to delete all properties.

Parameter	Description
name	The name of the property, or properties, to be deleted.

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

This method duplicates the specified object.

Parameter	Description
ctx	The object to be duplicated.

```
CORBA::Status get_default_context(CORBA::Context_ptr& ctx)
```

This method returns the default per-process `Context` maintained by ISB for C++. The default Context is often used in constructing DII requests. See Request for more information.

Parameter	Description
ctx	The context.

```
Status get_values(const char *start_scope,
                  Flags flags,
                  const char *name,
                  NVList_ptr& nvl) const
```

This method searches the Context object hierarchy and retrieves one or more of the name/value pairs specified by the `name` parameter. It then creates an `NVList` object, places the name/value pairs in the `NVList` and returns a reference to that object.

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

Parameter	Description
start_scope	The name of the `Context` object at which to start the search. If omitted, the search begins with this object. The search scope can be restricted to just this object by specifying `CORBA::CTX_RESTRICT_SCOPE`.
flags	An exception is raised if no matching context name is found.
name	The property name to search for. A trailing "*" wildcard character may be used to retrieve all properties that match `name`.
nvl	A reference to the list of properties found.

```
static Boolean CORBA::is_nil(CORBA::Context_ptr ctx)
```

This method checks the specified pointer and returns true if the pointer is `NULL`.

Parameter	Description
ctx	The object to be checked for a `NULL` value.

```
static Context_ptr _nil()
```

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

```
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
ctx	The object to be released.

```
static void CORBA::release(CORBA::Context_ptr ctx)
```

This method releases the specified object.

Parameter	Description
ctx	The object to be released.

```
Status set_one_value(const char *name, const Any& val)
```

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

Parameter	Description
name	The property's name.
val	The property's value.

```
Status set_values(NVList_ptr nvl)
```

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, the `Flags` field must be set to zero and each `Any` object added to the `NVList` must set its `TypeCode` to `TC_string`.

Parameter	Description
nvl	A list of name/value pairs to be added to this object.

CreationImplDef

```
class CreationImplDef : public ImplementationDef
```

The `CreationImplDef` class holds the all the information required to create an ORB object. This information is stored in the implementation repository when the object is registered with the **regobj** command or created using the `BOA::create` method. This class can also be used with the `BOA::change_implementation` method to alter an implementation definition that is already registered with the OAD. For more information, see "oad"

`CreationImplDef` adds these data members to those of ImplementationDef class.

Data member	Description
_path_name	The Path of the object implementation's executable. This is the file the Object Activation Daemon will start when a client requests the object.
_policy	The activation policy for the object implementation. See regobj Server Activation Policies for more information.
_args	Arguments to be passed to the server when it is activated.
_env	Environment variables to be set when the server is activated.

Include File

Include `corba.h` when using this class.

IDL Definition

```
interface CreationImplDef: ImplementationDef
{
  attribute string          path_name;
  attribute Policy          activation_policy;
  attribute StringSequence  args;
  attribute StringSequence  env;
};
```

Methods

- activation_policy
- args
- CreationImplDef
- _duplicate

- env
- _narrow
- _nil
- path_name
- _release

```
Policy activation_policy() const
```

This method returns this object's activation policy.

```
void activation_policy(Policy p)
```

This method sets this object's activation policy.

Parameter	Description
p	The activation policy to be set.

```
StringSequence *args() const
```

This method returns a list of the arguments that will be passed to the server when it is started.

```
void args(const StringSequence& val)
```

This method sets the list of the arguments that will be passed to the server when it is started.

Parameter	Description
val	The list of arguments to be passed.

```
CreationImplDef(const char * interface_name,
                const char * object_name,
                const ReferenceData& id,
                const char * path_name,
                const StringSequence& args,
                const StringSequence& env)
```

This method creates a `CreationImplDef` object.

Parameter	Description
interface_name	The interface name for the object implementation.
object_name	The object name.
id	Opaque data passed to the server. Reference data is often used to distinguish between different servers that have the same name.
path_name	The server's executable file.
args	A list of arguments to be passed to the server when it is started.
env	A list of environment variables to be set for the server when it is started.

```
static void CreationImplDef_ptr _duplicate(CreationImplDef_ptr obj)
```

This method duplicates the specified object.

Parameter	Description
obj	The object to be duplicated.

```
StringSequence *env() const
```

This method returns a list of the environment variables that will be set when the server is started.

```
void env(const StringSequence& val)
```

This method sets the list of environment variables that will be passed to the server when it is started.

Parameter	Description
val	The list of environment variables to be set.

```
static CreationImplDef_ptr _narrow(ImplementationDef_ptr ptr)
```

This method attempts to narrow the specified pointer to a `CreationImplDef_ptr`. If the method succeeds, a valid `CreationImplDef_ptr` is returned. Otherwise, a `NULL` pointer is returned.

Parameter	Description
ptr	The pointer to be narrowed to type `CreationImplDef_ptr`.

```
static CreationImplDef_ptr _nil()
```

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

```
const char *path_name() const
```

This method returns this object's path name.

```
void path_name(const char *val)
```

This method set this object's path name.

Parameter	Description
val	The path name to be set.

```
static void _release(CreationImplDef_ptr obj)
```

This method releases the specified object. Once the object's reference count reaches zero, the object will be deleted.

Parameter	Description
obj	The object to be released.

Environment

```
class 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, 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

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

Methods

- clear
- create_environment
- current_environment
- Environment
- ~Environment
- exception
- release
- void `clear`()

This method will cause this `Environment` to delete any `Exception` object that it holds. If this object holds no exception, this method has no effect.

```
Status ORB::create_environment(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
ptr	The pointer will be set to point to the newly created object.

```
static 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.

```
Environment()
```

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

```
~Environment()
```

This method deletes this `Environment` object. This is equivalent to calling the CORBA::`release` method, passing a pointer to the object to be deleted.

```
void exception(Exception *exp)
```

This method records the `Exception` object passed as an argument. The `Exception` object must be dynamically allocated because this 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
exp	A pointer to a dynamically allocated `Exception` object to be recorded for this `Environment`.

```
Exception *exception() const
```

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

```
static void CORBA::release(Environment_ptr env)
```

This method deletes the specified object, along with any `Exception` object it holds.

Parameter	Description
env	The object to be released.

Exception

```
class Exception
```

The `Exception` class is the base class of the SystemException and user exception classes. For more information, see SystemException in this guide.

Include File

Include `corba.h` when using this class.

HandlerRegistry

```
class HandlerRegistry
```

The `HandlerRegistry` class provides methods for registering both client and implementation-side event handlers with the ORB and the BOA. ISB for C++ provides two event handler classes, ClientEventHandler and ImplEventHandler, that define methods to be called by the ORB or BOA when certain events occur, such as a successful bind.

Event handlers can be registered with either a per-object scope or a global scope. If both a global event handler and an object event handler are registered, the per-object event handler will take precedence over the global handler for a particular object.

Event handlers are usually registered before any objects are bound or activated. If changes are made to a global event handler after an object is bound, the changes will not apply to event handling for that object.

To use the `HandlerRegistry`, you first obtain a reference using the static method `instance`.

Include File

Include the files `pcmext.h` and `corba.h` when using this class.

Methods

- instance
- reg_global_client_handler
- reg_global_impl_handler
- reg_obj_client_handler
- reg_obj_impl_handler
- unreg_glob_client_handler
- unreg_glob_impl_handler
- unreg_obj_client_handler
- unreg_obj_impl_handler

```
static HandlerRegistery_ptr instance()
```

This method returns a pointer to the `HandlerRegistry`.

```
void reg_global_client_handler(PMC_EXT::ClientEventHandler_ptr handler)
```

This method registers a global client-side event handler. The event handler will apply to any object used by the client. If a global event handler has already been registered, a `HandlerExists` exception will be raised.

Parameter	**Description**
handler	A pointer to the `ClientEventHandler` to be registered.

```
void reg_global_impl_handler(PMC_EXT::ImplEventHandler_ptr handler)
```

This method registers a global implementation-side event handler. The event handler will apply to any object used by the implementation. If a global event handler has already been registered, a `HandlerExists` exception will be raised.

Parameter	Description
handler	A pointer to the `ImplEventHandler` to be registered.

```
void reg_obj_client_handler(CORBA::OBject_ptr obj,
                            PMC_EXT::ClientEventHandler_ptr handler)
```

This method registers a client-side event handler for a particular object. The event handler will apply to any object used by the client. If an event handler has already been registered for this object, a `HandlerExists` exception will be raised. An `InvalidObject` exception will be raised if the object pointer is not valid.

Parameter	Description
obj	A pointer to the object for which this handler is being registered.
handler	A pointer to the `ClientEventHandler` to be registered.

```
void reg_obj_impl_handler(CORBA::OBject_ptr obj,
                          PMC_EXT::ImplEventHandler_ptr handler)
```

This method registers a implementation-side event handler for a particular object. The event handler will apply to any object used by this implementation. If an event handler has already been registered for this object, a `HandlerExists` exception will be raised. An `InvalidObject` exception will be raised if the object pointer is not valid.

Parameter	Description
obj	The object for which this event handler is to be registered.
handler	A pointer to the `ImplEventHandler` to be registered.

```
void unreg_glob_client_handler()
```

This method unregisters a global event handler for a client. If there is no global event handler currently registered, a `NoHandler` exception will be raised.

```
void unreg_glob_impl_handler()
```

This method unregisters a global event handler for an object implementation. If there is no global event handler currently registered, a `NoHandler` exception will be raised.

```
void unreg_obj_client_handler(CORBA::Object_ptr obj)
```

This method unregisters an client event handler for a particular object. If no event handler is currently registered for the specified object, a `NoHandler` exception will be raised. An `InvalidObject` exception will be raised if the object pointer is not valid.

Parameter	Description
obj	The object whose event handler is to be removed.

```
void unreg_obj_impl_handler(CORBA::Object_ptr obj)
```

This method unregisters an implementation event handler for a particular object. If no event handler is currently registered for the specified object, a `NoHandler` exception will be raised. An `InvalidObject` exception will be raised if the object pointer is not valid.

Parameter	Description
obj	The object whose event handler is to be removed.

IDLType

```
class IDLType : public IRObject
```

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

Include `corba.h` when using this class.

IDL Definition

```
interface IDLType:IRObject {
    readonly attribute TypeCode type;
};
```

Methods

- type

```
CORBA::Typecode_ptr type()
```

This method returns this object's typecode.

ImplementationDef

```
class ImplementationDef
```

The `ImplementationDef` is a class that contains most of the information needed to create or activate an object implementation.The `ActivationImplDef` class and the `CreationImplDef` class are both derived from this class.

Include File

Include `corba.h` when using this class.

IDL Definition

```
module CORBA {
  typedef sequence<string>       StringSequence;
  typedef sequence<octet>        OctetSequence;
  typedef OctetSequence          ReferenceData;
  interface ImplementationDef {
    attribute string interface_name;
    attribute string object_name;
    attribute ReferenceData id;
```

```
};
enum Policy {
  SHARED_SERVER,
  UNSHARED_SERVER,
  SERVER_PER_METHOD
};
```

Methods

- change_implementation
- _duplicate
- id
- interface_name
- _nil
- object_name
- release

```
void BOA::change_implementation(const object& obj,
                                const ImplementationDef& impl)
```

This BOA method changes the implementation definition for the specified object.

Parameter	Description
obj	The object whose definition is to be changed.
impl	The new implementation definition.

> **Caution:** Changing the implementation definition of an object after it has been created may make it inaccessible to existing client applications.

```
static ImplementationDef_ptr _duplicate(ImplementationDef_ptr obj)
```

This static method duplicates the specified pointer and returns the duplicated pointer.

Parameter	Description
obj	A pointer to the object to be duplicated.

```
ReferenceData_ptr *id() const
```

This method returns the `ReferenceData` associated with this object.

```
void id(const ReferenceData& *data)
```

This method sets the reference data for this object.

Parameter	Description
data	The reference data for this object.

```
const char *interface_name() const
```

This method returns the interface name associated with this object.

```
void interface_name(const char *val)
```

This method sets the interface name for this object.

Parameter	Description
val	The interface name for this object.

```
static ImplementationDef_ptr _nil()
```

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

```
const char *object_name() const
```

This method returns the object name associated with this object.

```
void object_name(const char *val)
```

This method sets the object name for this object.

Parameter	Description
val	The interface name for this object.

```
static void release(ImplementationDef_ptr obj)
```

This static method releases the specified pointer.

Parameter	Description
obj	The pointer to be released.

ImplEventHandler

```
class ImplEventHandler
```

The `ImplEventHandler` class defines the interface for registering an event handler for object implementations. Event handlers provide a set of methods that are invoked by the BOA when one of the following events occurs.

- A bind to an object is requested by a client.
- A client requests to be unbound from an object.
- A client aborts and the connection is lost.
- A client has requested the invocation of an object method.
- A method invocation requested by a client has completed.

You can derive your own event handling class from `ImplEventHandler` and provide the implementation just for the event handling methods in which you are interested. Your implementation will be invoked by the BOA when a particular event occurs and can be used to implement logging or security features that you design. If you are not interested in handling a particular event, do not declare its method in your derived class.

After creating an event handler, your object implementation must register it using the `HandlerRegistry` class. You can register an event handler for a particular object or a global event handler for all objects.

If both global and object-specific event handlers are registered and an event occurs on the object with its own event handler, the object-specific handler will take precedence over the global handler. If a global event handler is modified, the modifications will not apply to those object that were bound before the handler was modified.

For information on creating event handlers for client applications, see ClientEventHandler in this guide.

Include File

Include `corba.h` and `pmcext.h` when you use this class.

Methods

- bind
- client_aborted
- pre_method
- unbind

```
virtual void bind(const ConnectionInfo&,
                  CORBA::Principal_ptr,
                  CORBA::Object_ptr)
```

You can choose to implement this method, which will be invoked by the BOA when a client wants to bind to an object. If the event handler was registered for a particular object, it will only be invoked when a bind is requested for that object. If the event handler is registered with a global scope, it will be called when any bind request is received.

Parameter	**Description**
ConnectionInfo	This structure contains the host name, port, and file descriptor. You can use this information to adjust the characteristics of the connection.
CORBA::Principal_ptr	A pointer to `Principal` data associated with the client.
CORBA::Object_ptr	A pointer to the object the client has requested to be bound.

```
virtual void client_aborted(const ConnectionInfo&, CORBA::Object_ptr)
```

You can choose to implement this method, which will be invoked by the BOA when client aborts. If the event handler was registered for a particular object, it will only be invoked when a client that was bound to that object aborts. If the event handler is registered with a global scope, it will be called when any aborts.

Parameter	Description
ConnectionInfo	This structure contains the host name, port, and file descriptor.
CORBA::Object_ptr	A pointer to the object the client has requested to be bound.

```
virtual void pre_method(const ConnectionInfo&, CORBA::Principal_ptr,
                        const char *, CORBA::Object_ptr)
```

You can choose to implement this method, which will be invoked by the BOA before a client request for a method on an object is processed. After this method returns, the client's request is processed. If the event handler was registered for a particular object, it will only be invoked when a method is requested for that object. If the event handler is registered with a global scope, it will be called when an method is requested on any object.

Parameter	Description
ConnectionInfo	This structure contains the host name, port, and file descriptor. You can use this information to adjust the characteristics of the connection.
CORBA::Principal_ptr	A pointer to `Principal` object associated with the client.
char *	A pointer to the name of the operation.
CORBA::Object_ptr	A pointer to the object that implements the requested method.

```
virtual void pre_method(const ConnectionInfo&, CORBA::Principal_ptr,
                        const char *, CORBA::Object_ptr)
```

You can choose to implement this method, which will be invoked by the BOA after an object method has been invoked but before the results are returned to the client. If the event handler was registered for a particular object, it will only

be invoked after a method requested for that object has been processed. If the event handler is registered with a global scope, it will be called after any method is invoked on any object.

Parameter	Description
ConnectionInfo	This structure contains the host name, port, and file descriptor. You can use this information to adjust the characteristics of the connection.
CORBA::Principal_ptr	A pointer to `Principal` object associated with the client.
char *	A pointer to the name of the operation.
CORBA::Object_ptr	A pointer to the object that implements the requested method.

```
virtual void unbind(const ConnectionInfo&, CORBA::Object_ptr)
```

You can choose to implement this method, which will be invoked by the BOA when client requests to be unbound from an object. If the event handler was registered for a particular object, it will only be invoked when an unbind is requested for that object. If the event handler is registered with a global scope, it will be called when any unbind request is received.

Parameter	Description
ConnectionInfo	This structure contains the host name, port, and file descriptor. You can use this information to adjust the characteristics of the connection.
CORBA::Object_ptr	A pointer to the object the client has requested to be unbound.

InterfaceDef

```
class InterfaceDef : public Container, public Contained, public 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, Contained and IDLType in this guide.

Include File

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

IDL Definition

```
interface InterfaceDef: Container, Contained, IDLType {
   typedef sequence<RepositoryId> RepositoryIdSeq;
   typedef sequence<OperationDescription> OpDescriptionSeq;
   typedef sequence<AttributeDescription> AttrDescriptionSeq;
   attribute InterfaceDefSeq base_interfaces;
   boolean is_a(in RepositoryId interface_id);
   struct FullInterfaceDescription {
     Identifier name;
     RepositoryId id;
     RepositoryId defined_in;
     VersionSpec version;
     OpDescriptionSeq operations;
     AttrDescriptionSeq attributes;
     RepositoryIdSeq base_interfaces;
     TypeCode type;
   };
   FullInterfaceDescription describe_interface();
   AttributeDef create_attribute(
     in RepositoryId id,
     in Identifier name,
     in VersionSpec version,
     in IDLType type,
     in CORBA::AttributeMode mode
   );
   OperationDef create_operation(
     in RepositoryId id,
     in Identifier name,
     in VersionSpec 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;
  VersionSpec version;
```

```
  RepositoryIdSeq base_interfaces;
};
```

Methods

- base_interfaces
- create_attribute
- create_operation
- describe_interface
- is_a

```
InterfaceDefSeq *base_interfaces()
```

This method returns a list of the interfaces from which this class inherits.

```
void base_interfaces(const InterfaceDefSeq& val)
```

This method sets the list of the interfaces from which this class inherits. An error is returned if the name attribute of any object contained by this interface conflicts with the name attribute of any object contained by any of the interfaces specified in the sequence.

Parameter	Description
val	The list of interfaces from which this interface inherits.

```
AttributeDef_ptr create_attribute(const char * id,
                                  const char * name,
                                  const VersionSpec& version,
                                  IDLType_ptr type,
                                  AttributeMode mode)
```

This method returns a pointer to a newly created AttributeDef that is contained this object. The `id`, `name`, `version`, `type` and `mode` are set to the specified values. An error is returned if an object with the specified `id` already exists within the object repository or if an object with the specified `name` already exists within this `InterfaceDef`.

Parameter	Description
id	The interface id to use.
name	The interface name to use.
version	The interface version to use.
mode	The interface mode: readonly or normal.

```
OperationDef_ptr create_operation(const char *id,
                                  const char *name,
                                  IDLType_ptr result,
                                  OperationMode mode,
                                  const ParDescriptionSeq& params,
                                  const ExceptionDefSeq& exceptions,
                                  const ContextIdSeq& contexts)
```

This method creates an `OperationDef` that is contained by this object, using the specified parameters. The `defined_in` attribute of the `OperationDef` is set to identify this `InterfaceDef`.

Parameter	Description
id	The interface id for this operation.
name	The name of this operation.
result	The result type returned by this operation.
mode	The mode of this operation--oneway 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.

```
InterfaceDef::FullInterfaceDescription *describe_interface()
```

This method returns a FullInterfaceDescription that describes this object's interface.

```
Boolean is_a(const char * interface_id)
```

This method returns true if this interface is identical to or inherits from, directly or indirectly, from the specified interface.

Parameter	Description
interface_id	The id of the interface to be checked against this interface.

IRObject

```
class IRObject
```

The `IRObject` class offers the most generic interface for Interface Repository (IR) objects. The `Container` class, IDLType, Contained, and others are derived from this class.

Include File

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

IDL Definition

```
interface IRObject {
  readonly attribute DefinitionKind def_kind;
  void destroy();
};
```

Methods

- def_kind
- destroy

```
CORBA::DefinitionKind IRObject::def_kind()
```

This method returns the type of an Interface Repository object as one of the following `DefinitionKind` enumeration values: dk_Attribute, dk_Constant, dk_Exception, dk_Interface, dk_Module, dk_Operation, dk_Typedef, dk_Alias, dk_Struct, dk_Union, dk_Enum, dk_Primitive, dk_String, dk_Sequence, dk_Array, or dk_Repository.

```
void IRObject::destroy()
```

This method deletes this object. If this object is a `Container`, all of its contents will also be deleted. If the object is currently contained by another object, it will be removed. The `destroy` method cannot be invoked on a `Repository` object or on a `PrimitiveDef` object.

NamedValue

```
class 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. The `Any` class is used to represent the value associated with this object. See also the `Request` class.

Include File

Include `corba.h` when using this class.

Methods

- flags
- name
- value

```
Flags flags() const
```

This method returns the flag defining how this name-value pair is to be used. One of the following is returned.

- ARG_IN - This object represents an input parameter.
- ARG_OUT - This object represents an output parameter.
- ARG_INOUT - This object represents both an input and output parameter.
- IN_COPY_VALUE - This value can be specified in combination with the ARG_INOUT flag to specify that the ORB should make a copy of the 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.

```
Any *value() const
```

This method returns the value portion of this object's name-value pair.

NVList

```
class NVList
```

The NVList class is used to contain a list of NamedValue objects, and is used to pass parameters associated with a Dynamic Invocation Interface request. See also the Request class.

Include File

Include corba.h when using this class.

Methods

- add
- add_item
- add_value
- count

- _duplicate
- free_out_memory
- is_nil
- _nil
- release
- release
- remove

```
NamedValue_ptr add(Flags flags)
```

This method adds a NamedValue object to this list, initializing only the flags. Neither the name or value attribute of the added object are initialized. A pointer is returned which can be used to initialize the name and value attributes of the NamedValue.

Parameter	Description
flags	The flag indicating the intended use of the NamedValue object. It can be one of ARG_IN, ARG_OUT, or ARG_INOUT.

```
NamedValue_ptr add_item(const char *name, Flags flags)
```

This method adds a NamedValue object to this list, initializing both the flags and name attribute. A pointer is returned which can be used to initialize the value attribute of the NamedValue.

Parameter	Description
name	The name.
flags	The flag indicating the intended use of the NamedValue object. It can be one of ARG_IN, ARG_OUT, or ARG_INOUT.

```
NamedValue_ptr add_value(const char *name, const Any& val, Flags flags)
```

This method adds a NamedValue object to this list, initializing the name, value and flags. A pointer to the NamedValue object is returned.

Parameter	Description
name	The name.
val	The value.
flags	The flag indicating the intended use of the NamedValue object. It can be one of ARG_IN, ARG_OUT, or ARG_INOUT.

```
Long count() const
```

This method returns the number of NamedValue objects in this list.

```
static NVList_ptr _duplicate(NVlist_ptr ptr)
```

This static method duplicates the specified NVList_ptr.

Parameter	Description
ptr	The object to be duplicated.

```
Status free_out_memory()
```

This method releases all the memory associated with all output parameters.

```
static boolean CORBA::is_nil(NVList_ptr obj)
```

This method returns true if the specified NamedValue pointer is NULL.

Parameter	Description
obj	The object pointer to be checked.

```
NamedValue_ptr item(Long idx)
```

This method returns the NamedValue in the list with the specified index.

Parameter	Description
idx	The index of the desired NamedValue object. The index is zero-based.

```
static NamedValue_ptr _nil()
```

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

```
void _release()
```

This method releases this object.

```
static void CORBA::release(NVList_ptr obj)
```

This static method releases the specified object.

Parameter	Description
obj	The object to be released.

```
Status remove(Long idx)
```

This method deletes the NamedValue object from this list, located at the specified index.

Parameter	Description
idx	The index of the NamedValue object. The index is zero-based.

Object

```
class 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. The methods offered by the Object class are implemented by the ORB.

Include File

Include corba.h when using this class.

Methods

- Object Reference Methods
- Object State Methods

Object Reference Methods

- _duplicate
- _is_a
- _is_equivalent
- is_nil
- _is_persistent
- _nil
- _non_existent
- _ref_count
- _release

```
static Object_ptr _duplicate(Object_ptr obj)
```

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

Parameter	Description
obj	The object pointer to be duplicated.

```
Boolean _is_a(const char *repository_id)
```

This method returns true if this object implements the interface associated with the repository id. Otherwise, false is returned.

Parameter	Description
logical_type_id	The repository identifier to check.

```
Boolean _is_equivalent(Object_ptr other_object)
```

This method returns true if the specified object pointer points to this object. Otherwise, false is returned.

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

```
void CORBA::is_nil() const
```

This method returns true if this object's reference value is NULL. If it is not NULL, false is returned.

Note This method should not be used to check the return value from the `_narrow` method. Instead, compare it to `NULL`.

```
Boolean _is_persistent() const
```

This method returns true if this object is a persistent object. If this object is transient, false is returned.

```
static Object_ptr _nil()
```

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

```
Boolean _non_existent()
```

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

```
ULong _ref_count() const
```

This method returns the current reference count for this object. This is a local operation.

```
void _release()
```

This method releases this object and decrements its reference count. When an object's reference count reaches zero, the ORB will automatically delete the object.

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

This method releases this object and decrements its reference count. When an object's reference count reaches zero, the ORB will automatically delete the object.

Object State Methods

- _boa

- _create_request
- _default_principal
- _get_implementation
- _get_interface
- _hash
- _interface_name
- _is_local
- _is_remote
- _object_name
- _principal
- _request

```
BOA_ptr _boa() const
```

This method returns a pointer to the Basic Object Adaptor.

```
Status _create_request(Context_ptr ctx,
                       const char *operation,
                       NVList_ptr arg_list,
                       NamedValue_ptr result,
                       Request_ptr& request,
                       Flags req_flags)
```

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

Parameter	**Description**
ctx	The Context associated with this request. For more information, see Context.
operation	The name of the operation to be performed on the object implementation.

arg_list	A list of arguments to pass to the object implementation. See NVList for more information.
result	The result of the operation. See NamedValue for more information.
request	A pointer to the Request that is created. See Request for more information.
req_flags	This flag must be set to OUT_LIST_MEMORY if one or more of the `NamedValue` items in `arg_list is an output argument.`

```
static const Principal_ptr _default_principal()
```

This method returns a pointer to the default `Principal` for this object.

```
static void _default_principal(const Principal& principal) static const
```

This static method sets the global, default `Principal` for this object. For more information, see Principal.

Parameter	**Description**
principal	The default, global Principal to use for this object.

```
ImplementationDef_ptr _get_implementation()
```

This method returns a pointer to this object's implementation definition. See ImplementationDef for more information.

```
InterfaceDef_ptr _get_interface()
```

This method returns a pointer to this object's interface definition. See InterfaceDef for more information.

```
ULong _hash(ULong maximum)
```

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

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

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

This method returns this object's interface name.

```
Boolean _is_local() const
```

This method returns true if the object implementation resides on the same host as the client application.

```
Boolean _is_remote() const
```

This method returns true if the object implementation resides on a different host than the client application.

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

This method returns the object name associated with this object.

```
const Principal_ptr _principal() const
```

This method returns a pointer to the `Principal` associated with this object. For more information, see Principal.

```
void _principal(const Principal& principal)
```

This method sets the `Principal` for this object. For more information, see Principal.

Parameter	**Description**
principal	The Principal to use for this object. If the Principal is passed by reference, a new copy is created and used. If the Principal is passed as a pointer, that object will be used.

```
Request_ptr _request(Identifier operation)
```

This method creates a `Request` suitable for invoking methods on this objects. A pointer to the `Request` object is returned. See Request for more information.

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

OperationDef

```
class OperationDef : public Contained
```

The `OperationDef` class contains information about an interface operation that is stored in the Interface Repository. This class is derived from the Principal class. The inherited `describe` method returns a `OperationDescription` structure that provides complete information on the operation.

Include File

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

IDL Definition

```
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;
};
struct OperationDescription {
  Identifier name;
  RepositoryId id;
  RepositoryId defined_in;
  VersionSpec version;
  TypeCode result;
  OperationMode mode;
  ContextIdSeq contexts;
  ParDescriptionSeq parameters;
  ExcDescriptionSeq exceptions;
};
```

Methods

- context
- exceptions
- mode
- params
- result
- result_def

```
ContextIdSeq * context()
```

This method returns a list of context identifiers that apply to the operation.

```
void context(const ContextIdSeq& val)
```

This method sets the list of the context identifiers that apply to this operation.

Parameter	Description
val	The list of context identifiers.

```
ExceptionDefSeq * exceptions()
```

This method returns a list of the exception types that can be raised by this operation.

```
void exceptions(const 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.

```
OperationMode mode()
```

This method returns the mode of the operation represented by this `OperationDef`. The mode may be normal or oneway. Operations that have a normal mode are synchronous and return a value to the client application. Oneway operations do not block and no response is sent from the object implementation to the client.

```
void mode(OperationMode val)
```

This method sets the mode of the operation.

Parameter	Description
val	The desired mode of this operation, either OP_ONEWAY or OP_NORMAL.

```
ParDescriptionSeq * params()
```

This method returns a pointer to a list of `ParameterDescription` structures that describe the parameters to this `OperationDef`.

```
void params(const 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 `ParameterDescription` structures.

```
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.

```
IDLType_ptr result_def()
```

This method returns a pointer to the definition of the type returned by this `OperationDef`.

```
void result_def(IDLType_ptr val)
```

This method sets definition of the type returned by this `OperationDef`.

Parameter	Description
val	A pointer to the type definition to use.

ORB

```
class ORB
```

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

Include File

Include `corba.h` when using this class.

Methods

- create_alias_tc
- create_array_tc
- create_enum_tc
- create_environment
- create_exception_tc
- create_interface_tc
- create_list
- create_operation_list
- create_recursive_sequence_tc
- create_sequence_tc
- create_string_tc
- create_struct_tc
- create_union_tc
- _duplicate

- get_default_context
- get_next_response
- _nil
- object_to_string
- ORB_init
- poll_next_response
- release
- send_multiple_requests_oneway
- send_multiple_requests_deferred
- string_to_object

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

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

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

```
static TypeCode_ptr create_array_tc(Ulong bound, TypeCode_ptr
element_type)
```

This static method dynamically creates a TypeCode for an array.

Parameter	Description
bound	The maximum number of array elements.
element_type	The type of elements stored in this array.

```
static TypeCode_ptr create_enum_tc(const char *repository_id,
                                   const char *type_name,
                                   const EnumemberSeq& members)
```

This static method dynamically creates a TypeCode for an enumeration 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 enumeration's type.
members	A list of values for the enumeration's members.

```
Status create_environment(Enviranment_ptr& env)
```

This method creates an Environment object and returns a reference to the created object.

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

```
static TypeCode_ptr create_exception_tc(const char *repository_id,
                                        const char *type_name,
                                        const 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 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.

```
Status create_list(Long num, NVList_ptr& 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.

```
Status create_operation_list(OperationDef_ptr op, NVList& nvlist)
```

This method creates an argument list for the specified operation

Parameter	Description
op	Pointer to the operation definition whose argument list is to be created.
nvlist	A reference to the newly created argument list.

```
static TypeCode_ptr create_recursive_sequence_tc(Ulong bound,
                                                 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.

Parameter	Description
bound	The maximum number of sequence elements.
offset	The offset parameter determines which enclosing TypeCode describes the elements of this sequence.

```
static TypeCode_ptr create_sequence_tc(Ulong bound,
                                       TypeCode_ptr element_type)
```

This static method dynamically creates a TypeCode for a sequence.

Parameter	Description
bound	The maximum number of sequence elements.
element_type	The type of elements stored in this sequence.

```
static TypeCode_ptr create_struct_tc(const char *repository_id,
                                     const char *type_name,
                                     const StructMemberSeq& members)
```

This static method dynamically creates a TypeCode for the structure 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 TypeCode_ptr create_union_tc(const char *repository_id,
                                    const char *type_name,
                                    TypeCode_ptr discriminator_type,
                                    const UnionMemberSeq& members)
```

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

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

```
static TypeCode_ptr create_string_tc(Ulong bound)
```

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

Parameter	Description
bound	The maximum length of the string.

```
static ORB_ptr _duplicate(ORB_ptr ptr)
```

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

Parameter	Description
ptr	The ORB pointer to be duplicated.

```
Status get_default_context(Context_ptr& ctx)
```

This method returns a reference to the default `Context` for this process..

Parameter	Description
ctx	Initialized to the default Context for this process.

```
Status get_next_response(RequestSeq*& 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 calling this method.

Parameter	Description
req	Set to point to the request that has been received.

```
static ORB_ptr _nil()
```

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

```
char *object_to_string(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 etc. This is an ORB method because different ORB implementations may have different conventions for representing object references as strings.

Note The reference can be made persistent by saving it to a file; the object itself is not made persistent.

Parameter	Description
obj	Pointer to an object that is to be converted to a string.

```
static ORB_ptr ORB_init(int& argc,
                        char *const *argv,
                        const char *orb_id = "Internet ORB")
```

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. The arguments accepted by this method take the form of name-value pairs which allows them to be distinguished from other command line arguments.

Parameter	Description
argc	The number of arguments passed.
argv	The list of arguments passed.
orb_id	Identifies the type of ORB. Currently "Internet ORB" is the only supported value.

Name-Value pair	Description
-ORBagentaddr <ip_address>	Specifies the IP address of the osagent to be used. If this is not specified, an osagent will be located through the use of a broadcast message.
-ORBagentport <port_number>	Used, in conjunction with the above parameter, to specify the port number of the osagent to be used.
-ORBsendbufsize <size>	Specifies the size of the send buffer to be used by the network transport mechanism. If not specified, an appropriate default size will be used.
-ORBrcvbufsize <size>	Specifies the size of the receive buffer to be used by the network transport mechanism. If not specified, an appropriate default size will be used.

-ORBmbufsize <size>	Specifies the size of the intermediate buffer used by the ORB. If not specified, the ORB will maintain a pointer to the argument and will not make an intermediate copy. Using this parameter incorrectly can seriously affect performance.
-OAnoshm	Disables the use of shared memory for sending and receiving messages when the client and object implementation are located on the same host.
-OAshm	Enables the use of shared memory.
-ORBshmsize <size>	Specifies the size of the shared memory buffer used by the ORB. If this is not specified, an appropriate size will be used.

```
Boolean poll_next_response()
```

This method returns true if a response to a deferred request has been received, otherwise false is returned. This call does not block.

```
static void CORBA::release(ORB_ptr ptr)
```

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

Parameter	**Description**
prt	A pointer to the object to be released.

```
Status send_multiple_requests_deferred(const RequestSeq& req)
```

This method sends all the client request 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**
req	A sequence of deferred requests to be sent.

```
Status send_multiple_requests_oneway(const RequestSeq& req)
```

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

Parameter	Description
req	A sequence of oneway requests to be sent.

```
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
str	A pointer to a string representing an object.

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 client's `Principal`.

Include File

Include `corba.h` when using this typedef.

Methods

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

Repository

```
class Repository : public Container
```

The `Repository` class provides access to the interface repository and is derived from the `Container` class.

Include File

Include `corba.h` when using this class.

IDL Definition

```
interface CORBA_Repository: CORBA_Container {
    Contained lookup_id(in RepositoryId search_id);
    PrimitiveDef get_primitive(in CORBA::PrimitiveKind kind);
    StringDef create_string(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
    );
};
```

Methods

- create_array
- create_sequence
- create_string
- get_primitive
- lookup_id

```
ArrayDef_ptr create_array(ULong length, IDLType_ptr element_type)
```

This method creates a new `ArrayDef` and returns a pointer to that object.

Parameter	Description
length	The maximum number of elements in the array. This value must be greater than zero.
element_type	The IDLType of the elements stored in the array.

```
SequenceDef_ptr create_sequence(CORBA::ULong bound,
                                IDLType_ptr element_type)
```

This method creates a new `SequenceDef` object and returns a pointer to that object.

Parameter	Description
bound	The maximum number of items in the sequence. This value must be greater than zero.
element_type	A pointer to the IDLType of the items stored in the sequence.

```
StringDef_ptr create_string(Ulong bound)
```

This method creates a new `StringDef` object and returns a pointer to that object.

Parameter	Description
bound	The maximum length of the string. This value must be greater than zero.

```
PrimitiveDef_ptr get_primitive(PrimitiveKind kind)
```

This method returns a reference to a PrimitiveKind.

Parameter	Description
kind	The reference returned to the

```
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
search_id	The identifier to use for the search.

Request

```
class 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` specifies an operation to be performed on the ORB object, 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.

The Object class provides the methods `_create_request` and `_request` for creating a `Request` object.

Include File

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

Methods

- arguments
- ctx
- env
- get_response
- invoke
- operation

- poll_response
- result
- send_deferred
- send_oneway
- target

```
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.

```
void ctx(CORBA::Context_ptr ctx)
```

This method sets the `Context` to be used with this request.

Parameter	Description
ctx	The Context object to be associated with this request.

```
CORBA::Context_ptr ctx() const
```

This method returns a pointer to the `Context` associated with this request.

```
CORBA::Environment_ptr env()
```

This method returns a pointer to the `Environment` associated with this request.

```
CORBA::Status 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.

```
CORBA::Status invoke()
```

This method invokes this `Request` on the ORB object associated with this request. This method will block 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 will represent.

```
CORBA::Boolean poll_response()
```

This nonblocking 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 false is returned.

```
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.

```
CORBA::Status 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.

```
CORBA::Status send_oneway()
```

This method invokes this `Request` as 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.

SystemException

```
class SystemException : public Exception
```

The `SystemException` class is used to report standard system errors encountered by the ORB or by the object implementation. This class is derived from the `Exception` class, which provides methods for printing the name and details of the exception to an output stream.

`SystemException` object include a completion status, that 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

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

Methods

- completed
- minor
- _narrow
- SystemException

```
CompletionStatus completed() const
```

This method returns true if this object's completion status is set to `COMPLETED_YES;` otherwise, it returns false.

```
void completed(CompletionStatus status)
```

This method sets the completion status for this object.

Parameter	Description
status	The completion status, one of COMPLETED_YES, COMPLETED_NO, or COMPLETED_MAYBE.

```
ULong minor() const
```

This method returns this object's minor code.

```
void minor(ULong val)
```

This method sets the minor code for this object.

Parameter	Description
val	The minor code.

```
static SystemException *_narrow(Exception *exc)
```

This method attempts to narrow 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**
exc	An Exception pointer to be narrowed.

SystemException(ULong **minor** = 0, completion_status **status** =

Exception Name	Description
UNKNOWN	Unknown exception.
BAD_PARAM	An invalid parameter was passed.
NO_MEMONY	Dynamic memory allocation failure.
IMP_LIMIT	Implementation limit violated.
COMM_FAILURE	Communication failure.
INV_OBJREF	Invalid object reference specified.
NO_PERMISSION	No permission for attempted operation.
INTERNAL	ORB internal error.
MARSHAL	Error marshalling parameter or result.
INITIALIZE	ORB initialization failure.
NO_IMPLEMENT	Operation implementation not available.
BAD_TYPECODE	Invalid typecode.
BAD_OPERATION	Invalid operation.
NO_RESOURCES	Insufficient resources to process request.
NO_RESPONSE	Response to request not yet available.
PERSIST_STORE	Persistent storage failure.
BAD_INV_ORDER	Routine invocations out of order.
TRANSIENT	Transient failure.
FREE_MEM	Unable to free memory.
INV_INDENT	Invalid identifier syntax.
INV_FLAG	Invalid flag was specified.
INTF_REPOS	Error accessing interface repository.
BAD_CONTEXT	Error processing context object.
OBJ_ADAPTOR	Failure detected by object adaptor.
DATA_CONVERSION	Data conversion error.
OBJECT_NOT_EXIST	Object is not available.

COMPLETED_NO)

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

Parameter	Description
minor	The minor code.
status	The completion status, one of COMPLETED_YES, COMPLETED_NO, or COMPLETED_MAYBE.