Contents

1. OMG IDL Syntax and the C++ IDL Compiler
   OMG IDL Compiler Extensions ......................................................... 1-2
   C++ IDL Compiler Constraints .................................................... 1-3

2. Implementation Configuration File (ICF)
   ICF Syntax .................................................................................. 2-2
   Sample ICF File ............................................................................ 2-3
   Creating the ICF File ..................................................................... 2-4
   See Also ....................................................................................... 2-4

3. TP Framework
   A Simple Programming Model ...................................................... 3-3
   Control Flow ................................................................................ 3-3
   Object State Management ............................................................. 3-4
   Transaction Integration ................................................................. 3-4
   Object Housekeeping .................................................................. 3-4
   High-level Services ..................................................................... 3-5
   State Management ....................................................................... 3-5
   Activation Policy ......................................................................... 3-5
   Application-controlled Activation and Deactivation ..................... 3-7
   Servant Lifetime ......................................................................... 3-10
   Saving and Restoring Object State .............................................. 3-12
   Transactions ............................................................................... 3-12
Transaction Policies .......................................................... 3-13
Transaction Initiation ....................................................... 3-14
Transaction Termination .................................................... 3-14
Transaction Suspend and Resume ........................................ 3-14
Restrictions on Transactions .............................................. 3-16
SQL and Global Transactions ............................................. 3-16
Voting on Transaction Outcome .......................................... 3-17
Transaction Timeouts ....................................................... 3-18
IIOP Client Failover .......................................................... 3-18
Setting The Retry Policy ................................................... 3-18
Initiating IIOP Client Failover ........................................... 3-19
See Also ............................................................................ 3-20
Weblogic CORBA Clustering and Load Balancing Support .......... 3-20
Parallel Objects ................................................................. 3-20
TP Framework API ............................................................. 3-22
Server Interface ............................................................... 3-23
ServerBase Interface .......................................................... 3-24
Server::create_servant() ...................................................... 3-25
ServerBase::create_servant_with_id() .................................. 3-27
Server::initialize() ............................................................ 3-29
ServerBase::thread_initialize() .......................................... 3-31
Server::release() .............................................................. 3-33
ServerBase::thread_release() .............................................. 3-35
Tobj_ServantBase Interface .................................................. 3-36
Tobj_ServantBase::activate_object() .................................... 3-37
Tobj_ServantBase::_add_ref() ............................................. 3-39
Tobj_ServantBase::deactivate_object() .................................. 3-40
Tobj_ServantBase::_is_reentrant() ...................................... 3-45
Tobj_ServantBase::_remove_ref() ................................. 3-46
TP Interface ....................................................... 3-46
  TP::application_responsibility() .............................. 3-48
  TP::bootstrap() ............................................... 3-49
  TP::close_xa_rm() ............................................ 3-50
  TP::create_active_object_reference() ......................... 3-51
  TP::create_object_reference() ................................ 3-54
  TP::deactivateEnable() ....................................... 3-56
  TP::get_object_id () .......................................... 3-58
  TP::get_object_reference() .................................... 3-59
  TP::open_xa_rm() .............................................. 3-60
  TP::orb() ...................................................... 3-61
  TP::register_factory() ........................................ 3-62
  TP::unregister_factory() ...................................... 3-63
  TP::userlog() .................................................. 3-65

CosTransactions::TransactionalObject Interface Not Enforced ............... 3-66

Error Conditions, Exceptions, and Error Messages ................................ 3-66
Exceptions Raised by the TP Framework ........................................... 3-66
Exceptions in the Server Application Code ....................................... 3-66
Exceptions and Transactions ....................................................... 3-67
Restriction of Nested Calls on CORBA Objects ................................... 3-67

4. CORBA Bootstrapping Programming Reference

  Why Bootstrapping Is Needed ........................................ 4-2
  Supported Bootstrapping Mechanisms .................................... 4-2
  BEA Bootstrapping Mechanism ......................................... 4-2
    How Bootstrap Objects Work ....................................... 4-2
  Types of BEA Remote Clients Supported .................................. 4-7
5. FactoryFinder Interface

Capabilities, Limitations, and Requirements .................................................. 5-2
Functional Description ..................................................................................... 5-2
Locating a FactoryFinder .................................................................................. 5-3
Registering a Factory ....................................................................................... 5-3
Locating a Factory ............................................................................................ 5-5
Creating Application Factory Keys ................................................................. 5-10

C++ Member Functions and Java Methods ....................................................... 5-18
   CosLifeCycle::FactoryFinder::find_factories .............................................. 5-18
   Tobj::FactoryFinder::find_one_factory ...................................................... 5-20
   Tobj::FactoryFinder::find_one_factory_by_id ............................................ 5-22
   Tobj::FactoryFinder::find_factories_by_id ............................................... 5-24
   Tobj::FactoryFinder::list_factories ......................................................... 5-26
Automation Methods ....................................................................................... 5-27
   DIITobj_FactoryFinder.find_one_factory ............................................... 5-27
   DIITobj_FactoryFinder.find_one_factory_by_id ........................................ 5-28
   DIITobj_FactoryFinder.find_factories_by_id ............................................ 5-30
   DIITobj_FactoryFinder.find_factories ..................................................... 5-31
   DIITobj_FactoryFinder.list_factories ..................................................... 5-32

Programming Examples ................................................................................... 5-33
   Using the FactoryFinder Object ............................................................. 5-34
   Using Extensions to the FactoryFinder Object ....................................... 5-36
6. Security Service
7. Transactions Service
8. Notification Service
9. Request-Level Interceptors
10. CORBA Interface Repository Interfaces

Structure and Usage ........................................ 10-2
Programming Information ................................ 10-3
Performance Implications ............................... 10-4
Building Client Applications ......................... 10-4
Getting Initial References to the InterfaceRepository Object ........ 10-5
Interface Repository Interfaces ......................... 10-5
Supporting Type Definitions ......................... 10-5
IRObj ect Interface .................................... 10-6
Contained Interface .................................. 10-7
Container Interface .................................. 10-8
IDLType Interface .................................... 10-10
Repository Interface ................................. 10-11
ModuleDef Interface ................................ 10-11
ConstantDef Interface .............................. 10-12
TypedefDef Interface ............................... 10-13
StructDef ........................................... 10-13
UnionDef ............................................ 10-14
EnumDef ............................................. 10-15
AliasDef ............................................. 10-15
PrimitiveDef ......................................... 10-15
11. Joint Client/Servers

Introduction ......................................................... 11-2
Main Program and Server Initialization .............. 11-2
Servants .......................................................... 11-3
Servant Inheritance from Skeletons .................. 11-3
Callback Object Models Supported .................... 11-4
Configuring Servers to Call Remote Joint Client/Server Objects .......... 11-5
Preparing Callback Objects Using CORBA (C++ Joint Client/Servers Only) . . . 11-6
Preparing Callback Objects Using BEAWrapper Callbacks ................. 11-8

C++ BEAWrapper Callbacks Interface API .............. 11-11

Callbacks ....................................................... 11-11
start_transient .................................................. 11-12
start_persistent_systemid ................................. 11-13
restart_persistent_systemid ......................... 11-15
start_persistent_userid ................................. 11-17
stop_object .................................................. 11-19
stop_all_objects ............................................ 11-19
get_string_oid .............................................. 11-20
~Callbacks .................................................... 11-21
12. Development Commands

13. Mapping of OMG IDL Statements to C++

Mappings ................................................................. 13-1
  Data Types .......................................................... 13-2
  Strings ................................................................. 13-4
  wchar ................................................................. 13-5
  wstrings ............................................................ 13-5
  wstrings ............................................................ 13-5
  Constants ........................................................... 13-6
  Enums ................................................................. 13-7
  Structs .............................................................. 13-7
  Unions ............................................................... 13-9
  Sequences .......................................................... 13-15
  Arrays ............................................................... 13-19
  Exceptions ........................................................ 13-21
  Mapping of Pseudo-objects to C++ ................................. 13-23
  Usage ................................................................. 13-24
  Mapping Rules ....................................................... 13-24
  Relation to the C PIDL Mapping ................................. 13-25
  Typedefs ........................................................... 13-26
  Implementing Interfaces ........................................... 13-27
  Implementing Operations .......................................... 13-29
  PortableServer Functions ....................................... 13-31
  Modules ............................................................ 13-31
  Interfaces .......................................................... 13-32
  Generated Static Member Functions ............................. 13-33
  Object Reference Types ........................................... 13-34
  Attributes .......................................................... 13-34
14. CORBA API

Global Classes .................................................. 14-1
Pseudo-objects ................................................. 14-2
Any Class Member Functions ................................ 14-2

CORBA::Any::Any() ........................................... 14-3
CORBA::Any::Any(const CORBA::Any & InitAny) ........... 14-4
CORBA::Any::Any(TypeCode_ptr TC, void * Value, Boolean Release) ... 14-5
CORBA::Any::~Any() ........................................... 14-5
CORBA::Any & CORBA::Any::operator=(const CORBA::Any & InitAny) 14-6
void CORBA::any::operator <<=() .......................... 14-7
CORBA::Boolean CORBA::Any::operator >>=() ............... 14-7
CORBA::Any::operator <<=() .................................. 14-9
CORBA::Boolean CORBA::Any::operator >>=() ............... 14-9
CORBA::TypeCode_ptr CORBA::Any::type() const .......................... 14-10
void CORBA::Any::replace() .................................................. 14-11

Context Member Functions ................................................... 14-11

Memory Management .......................................................... 14-12
CORBA::Context::context_name ............................................ 14-12
CORBA::Context::create_child ............................................. 14-13
CORBA::Context::delete_values .......................................... 14-14
CORBA::Context::get_values ............................................... 14-15
CORBA::Context::parent ..................................................... 14-16
CORBA::Context::set_one_value ......................................... 14-17
CORBA::Context::set_values ............................................... 14-18

ContextList Member Functions .............................................. 14-18
CORBA::ContextList::count ............................................... 14-19
CORBA::ContextList::add ................................................. 14-20
CORBA::ContextList::add_consume ...................................... 14-21
CORBA::ContextList::item ................................................. 14-21
CORBA::ContextList::remove .............................................. 14-22

NamedValue Member Functions .............................................. 14-23
Memory Management .......................................................... 14-23
CORBA::NamedValue::flags ................................................. 14-24
CORBA::NamedValue::name ................................................... 14-24
CORBA::NamedValue::value .................................................. 14-25

NVList Member Functions .................................................... 14-26
Memory Management .......................................................... 14-26
CORBA::NVList::add .......................................................... 14-27
CORBA::NVList::add_item ................................................... 14-28
CORBA::NVList::add_value .................................................. 14-29
CORBA::NVList::count ........................................................ 14-30
<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>CORBA::NVList::item</td>
<td>14-31</td>
</tr>
<tr>
<td>CORBA::NVList::remove</td>
<td>14-32</td>
</tr>
<tr>
<td><strong>Object Member Functions</strong></td>
<td></td>
</tr>
<tr>
<td>CORBA::Object::_create_request</td>
<td>14-34</td>
</tr>
<tr>
<td>CORBA::Object::_duplicate</td>
<td>14-35</td>
</tr>
<tr>
<td>CORBA::Object::_get_interface</td>
<td>14-36</td>
</tr>
<tr>
<td>CORBA::Object::_is_a</td>
<td>14-37</td>
</tr>
<tr>
<td>CORBA::Object::_is_equivalent</td>
<td>14-38</td>
</tr>
<tr>
<td>CORBA::Object::_nil</td>
<td>14-38</td>
</tr>
<tr>
<td>CORBA::Object::_non_existent</td>
<td>14-39</td>
</tr>
<tr>
<td>CORBA::Object::_request</td>
<td>14-40</td>
</tr>
<tr>
<td><strong>CORBA Member Functions</strong></td>
<td></td>
</tr>
<tr>
<td>CORBA::release</td>
<td>14-41</td>
</tr>
<tr>
<td>CORBA::is_nil</td>
<td>14-42</td>
</tr>
<tr>
<td>CORBA::hash</td>
<td>14-43</td>
</tr>
<tr>
<td>CORBA::resolve_initial_references</td>
<td>14-44</td>
</tr>
<tr>
<td><strong>ORB Member Functions</strong></td>
<td></td>
</tr>
<tr>
<td>CORBA::ORB::clear_ctx</td>
<td>14-46</td>
</tr>
<tr>
<td>CORBA::ORB::create_context_list</td>
<td>14-47</td>
</tr>
<tr>
<td>CORBA::ORB::create_environment</td>
<td>14-47</td>
</tr>
<tr>
<td>CORBA::ORB::create_exception_list</td>
<td>14-48</td>
</tr>
<tr>
<td>CORBA::ORB::create_list</td>
<td>14-48</td>
</tr>
<tr>
<td>CORBA::ORB::create_named_value</td>
<td>14-49</td>
</tr>
<tr>
<td>CORBA::ORB::create_operation_list</td>
<td>14-50</td>
</tr>
<tr>
<td>CORBA::ORB::create_policy</td>
<td>14-51</td>
</tr>
<tr>
<td>CORBA::ORB::destroy</td>
<td>14-54</td>
</tr>
<tr>
<td>CORBA::ORB::get_ctx</td>
<td>14-54</td>
</tr>
<tr>
<td>CORBA::ORB::get_default_context</td>
<td>14-55</td>
</tr>
</tbody>
</table>
CORBA::ORB::get_next_response ................................. 14-56
CORBA::ORB::inform_thread_exit ............................... 14-57
CORBA::ORB::list_initial_services ............................. 14-57
CORBA::ORB::object_to_string ................................ 14-58
CORBA::ORB::perform_work .................................... 14-59
CORBA::ORB::poll_next_response ............................... 14-60
CORBA::ORB::resolve_initial_references ...................... 14-61
CORBA::ORB::send_multiple_requests_deferred ................. 14-62
CORBA::ORB::send_multiple_requests_oneway .................. 14-63
CORBA::ORB::set_ctx .......................................... 14-64
CORBA::ORB::string_to_object ................................ 14-64
CORBA::ORB::work_pending .................................... 14-65
ORB Initialization Member Function .......................... 14-66
CORBA::ORB_init ................................................. 14-67
ORB ............................................................. 14-69
Policy Member Functions ......................................... 14-74
CORBA::Policy::copy ............................................. 14-75
CORBA::Policy::destroy ......................................... 14-75
PortableServer Member Functions ............................... 14-76
PortableServer::POA::activate_object ......................... 14-77
PortableServer::POA::activate_object_with_id ............... 14-78
PortableServer::POA::create_id_assignment_policy .......... 14-79
PortableServer::POA::create_lifespan_policy ................ 14-80
PortableServer::POA::create_POA ................................ 14-81
PortableServer::POA::create_reference ....................... 14-83
PortableServer::POA::create_reference_with_id ............. 14-84
PortableServer::POA::deactivate_object ..................... 14-85
PortableServer::POA::destroy ................................. 14-86
PortableServer::POA::find_POA ........................................... 14-87
PortableServer::POA::reference_to_id ................................. 14-88
PortableServer::POA::the_POAManager ................................. 14-88
PortableServer::ServantBase::_default_POA .......................... 14-89

POA Current Member Functions ........................................ 14-90
PortableServer::Current::get_object_id .............................. 14-90
PortableServer::Current::get_POA .................................... 14-91

POAManager Member Functions ....................................... 14-91
PortableServer::POAManager::activate ................................ 14-92
PortableServer::POAManager::deactivate ............................. 14-93

POA Policy Member Objects .......................................... 14-94
PortableServer::LifespanPolicy ...................................... 14-94
PortableServer::IdAssignmentPolicy ................................. 14-95

Request Member Functions ........................................... 14-95
CORBA::Request::arguments ........................................... 14-96
CORBA::Request::ctx(Context_ptr) ................................. 14-97
CORBA::Request::get_response ...................................... 14-97
CORBA::Request::invoke .............................................. 14-98
CORBA::Request::operation .......................................... 14-99
CORBA::Request::poll_response ..................................... 14-99
CORBA::Request::result ............................................... 14-100
CORBA::Request::env .................................................. 14-100
CORBA::Request::ctx .................................................. 14-101
CORBA::Request::contexts .......................................... 14-102
CORBA::Request::exceptions ............................... .................. 14-102
CORBA::Request::target .............................................. 14-103
CORBA::Request::send_deferred ................................. 14-103
CORBA::Request::send_oneway ...................................... 14-104
15. Server-side Mapping

Implementing Interfaces .................................................. 15-1
Inheritance-based Interface Implementation .......................... 15-2
Delegation-based Interface Implementation .......................... 15-4
Implementing Operations .................................................. 15-8
The Object Management Group (OMG) Interface Definition Language (IDL) is used to describe the interfaces that client objects call and that object implementations provide. An OMG IDL interface definition fully specifies each operation’s parameters and provides the information needed to develop client applications that use the interface’s operations.

Client applications are written in languages for which mappings from OMG IDL statements have been defined. How an OMG IDL statement is mapped to a client language construct depends on the facilities available in the client language. For example, an OMG IDL exception might be mapped to a structure in a language that has no notion of exception, or to an exception in a language that does.

OMG IDL statements obey the same lexical rules as C++ statements, although new keywords are introduced to support distribution concepts. OMG IDL statements also provide full support for standard C++ preprocessing features and OMG IDL-specific pragmas.

**Note:** When using a pragma version statement, be sure to locate it after the corresponding interface definition. The following is an example of proper usage:

```c++
module A
{
    interface B
    {
        #pragma version B "3.5"
        void op1();
    };
};
```
The OMG IDL grammar is a subset of ANSI C++ with additional constructs to support the operation invocation mechanism. OMG IDL is a declarative language; it supports C++ syntax for constant, type, and operation declarations; it does not include any algorithmic structures or variables.

For a description of OMG IDL grammar, see Chapter 3 of the Common Object Request Broker: Architecture and Specification Revision 2.4 “OMG IDL Syntax and Semantics.”

All OMG IDL grammar is supported, with the exception of the following type declarations and associated literals:

- `native`

  **Note:** Because CORBA 2.4 states that the `native` type declaration is intended for use in Object Adapters, not user interfaces, this type is available in the `PortableServer` module only for clients that support callbacks, that is, joint client/servers.

- `long double`
- `fixed`

Do not use these data types in IDL definitions.

**Note:** Support for the `long long`, `unsigned long long`, `wchar`, and `wstring` data types was added to BEA Tuxedo CORBA in release 8.0.

### OMG IDL Compiler Extensions

The IDL compiler defines preprocessor macros specific to the platform. All macros predefined by the preprocessor that you are using can be used in the OMG IDL file, in addition to the user-defined macros. You can also define your own macros when you are compiling or loading OMG IDL files.

Table 1-1 describes the predefined macros for each platform.

<table>
<thead>
<tr>
<th>Macro Identifier</th>
<th>Platform on Which the Macro Is Defined</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>unix</strong></td>
<td>Sun Solaris, HP-UX, and IBM AIX</td>
</tr>
<tr>
<td><strong>sun</strong></td>
<td>Sun Solaris</td>
</tr>
<tr>
<td><strong>hpux</strong></td>
<td>HP-UX</td>
</tr>
</tbody>
</table>
C++ IDL Compiler Constraints

Table 1-1  Predefined Macros

<table>
<thead>
<tr>
<th>Macro Identifier</th>
<th>Platform on Which the Macro Is Defined</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>aix</strong></td>
<td>IBM AIX</td>
</tr>
<tr>
<td><strong>win_nt</strong></td>
<td>Microsoft Windows</td>
</tr>
</tbody>
</table>

C++ IDL Compiler Constraints

Table 1-2 describes constraints for the BEA Tuxedo 9.1 C++ IDL compiler and provides information about recommended workarounds.

Table 1-2  C++ IDL Compiler Constraints

<table>
<thead>
<tr>
<th>Constraint</th>
<th>Use of wildcarding in OMG IDL context strings produces warnings.</th>
</tr>
</thead>
</table>
| **Description** | A warning is generated by the C++ IDL compiler when context strings that contain wildcard characters are used in the operation definitions. When you specify a context string in an OMG IDL operation definition, the following warning may be generated:  
void op5() context("*");  
^  
LIBORBCMD_CAT:131: INFO: ‘*’ is a non-standard context property. |
| **Workaround** | The OMG CORBA specification is ambiguous about whether the first character of a context string must be alphabetic.  
This warning is generated to inform you that you are not in compliance with some interpretations of the OMG CORBA specification. If you are intending to specify all strings as context string values, as shown above, the OMG CORBA specification requires a comma-separated list of strings, in which the first character is alphabetic.  
**Note:** The example shown above is not OMG CORBA compliant, but it is processed by the BEA Tuxedo software as intended by the user. |

Constraint: Use of wildcarding in OMG IDL context strings produces warnings.
A warning is generated by the C++ IDL compiler when context strings that contain wildcard characters are used in the operation definitions. When you specify a context string in an OMG IDL operation definition, the following warning may be generated:

```cpp
void op5() context("*");
^ LIBORBCMD_CAT:131: INFO: '*' is a non-standard context property.
```

The OMG CORBA specification is ambiguous about whether the first character of a context string must be alphabetic. This warning is generated to inform you that you are not in compliance with some interpretations of the OMG CORBA specification. If you are intending to specify all strings as context string values, as shown above, the OMG CORBA specification requires a comma-separated list of strings, in which the first character is alphabetic.

**Note:** The example shown above is not OMG CORBA compliant, but it is processed by the BEA Tuxedo software as intended by the user.

<table>
<thead>
<tr>
<th>Constraint</th>
<th>The C++ IDL compiler does not support some data types.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>The C++ IDL compiler currently does not support the following data types, which are defined in the CORBA specification version 2.4:</td>
</tr>
<tr>
<td></td>
<td>• native</td>
</tr>
<tr>
<td></td>
<td>• fixed</td>
</tr>
<tr>
<td></td>
<td>• long double</td>
</tr>
<tr>
<td>Workaround</td>
<td>Avoid using these data types in IDL definitions.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Constraint</th>
<th>Using certain substrings in identifiers may cause incorrect code generation by the C++ IDL compiler.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>Using the following substrings in identifiers may cause code to be generated incorrectly and result in errors when the generated code is compiled:</td>
</tr>
<tr>
<td></td>
<td>get_ set_ Impl_ _ptr _slice</td>
</tr>
<tr>
<td>Workaround</td>
<td>Avoid the use of these substrings in identifiers.</td>
</tr>
<tr>
<td>Constraint</td>
<td>Inconsistent behavior in IDL compiler regarding case sensitivity.</td>
</tr>
<tr>
<td>-----------</td>
<td>-----------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>Description</strong></td>
<td>According to the CORBA standard, IDL identifiers that differ only in case should be considered colliding and yield a compilation error. There is a current limitation of the BEA Tuxedo IDL compiler for C++ bindings in that it does not always detect and report such name collisions except for value type. Value type will follow CORBA standard regarding case sensitivity.</td>
</tr>
<tr>
<td><strong>Workaround</strong></td>
<td>Avoid using IDL identifiers that differ only in case.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Constraint</th>
<th>C++ IDL typedef problem.</th>
</tr>
</thead>
</table>
| **Description** | The C++ IDL compiler generates code that does not compile when:  
• Defining IDL variables of `char` or `boolean` type  
• And the type is aliased multiple times  
For example, the generated C++ code from the following IDL code will not compile:  
```
module X {
    typedef boolean a;
    typedef a b;
    interface Y {
        attribute b Z;
    };
};
```
C++ compilers report an error that an "operator <<" is ambiguous and that there is no "operator>>" for type `char`. These errors are produced because of the multiple levels of typedefs; the C++ compiler may not associate the type `X::b` with `CORBA::Boolean` because of the intermediate type definition of `X::a`. |
| **Workaround** | Use a single level of indirection when you define `char` or `boolean` types. In the IDL example above, the attribute ‘`X::Z`’ would be defined using either the standard type ‘`boolean`’ or the user type ‘`X::a`’, but not the user type ‘`X::b`’. |
Implementation Configuration File (ICF)

The BEA Tuxedo CORBA TP Framework application programming interface (API) provides callback methods for object activation and deactivation. These methods provide the ability for application code to implement flexible state management schemes for CORBA objects.

State management is the way you control the saving and restoring of object state during object deactivation and activation. State management also affects the duration of object activation, which influences the performance of servers and their resource usage. The external API of the TP Framework includes the `activate_object()` and `deactivate_object()` methods, which provide a possible location for state management code. Additionally, the TP Framework API includes the `deactivateEnable()` method to enable the user to control the timing of object deactivation. The default duration of object activation is controlled by policies assigned to implementations at OMG IDL compile time.

While CORBA objects are active, their state is contained in a servant. This state must be initialized when objects are first invoked (that is, the first time a method is invoked on a CORBA object after its object reference is created) and on subsequent invocations after objects have been deactivated.

While a CORBA object is deactivated, its state must be saved outside the process in which the servant was active. When an object is activated, its state must be restored. The object’s state can be saved in shared memory, in a file, in a database, and so forth. It is up to the programmer to determine what constitutes an object’s state, and what must be saved before an object is deactivated and restored when an object is activated.

You can use the Implementation Configuration File (ICF) to set activation policies to control the duration of object activations in each implementation. The ICF file manages object state by
specifying the activation policy. The activation policy determines the in-memory activation duration for a CORBA object. A CORBA object is active in a Portable Object Adapter (POA) if the POA’s active object map contains an entry that associates an object ID with an existing servant. Object deactivation removes the association of an object ID with its active servant.

ICF Syntax

ICF syntax is as follows:

```plaintext
[#pragma activation_policy method|transaction|process]
[#pragma transaction_policy never|ignore|optional|always]
[#pragma concurrency_policy user_controlled|system_controlled]
[#pragma retry_policy never|always]
[Module module-name []
  implementation [implementation-name]
  {
    implements (module-name::interface-name);
    [activation_policy (method|transaction|process)];
    [transaction_policy (never|ignore|optional|always)];
    [concurrency_policy (user_controlled|system_controlled)];
    [retry_policy (never|always)];
  };[]];
```

pragmas

The four optional pragmas allow you to set a specific policy as the default policy for the entire ICF for all implementations that do not have an explicit activation_policy, transaction_policy, concurrency_policy, or retry_policy statement. This feature relieves the programmer from having to specify policies for each implementation and/or allows overriding of the defaults.

Module module-name

The module-name variable is optional if it is optional in the OMG IDL file. This variable is used for scoping and grouping. Its use must be consistent with the way it is used inside the OMG IDL file.

implementation-name

This variable is optional and is used as the name of the servant or as the class name in the server. It is constructed using interface-name with an _i appended if it is not specified by the programmer.

implements (module-name::interface-name)

This variable identifies the module and the interface to which the activation policy and the transaction policy apply.
activation_policy
For a description of the activation policies, see Activation Policy.

transaction_policy
For a description of the transaction policies, see Transaction Policies.

concurrency_policy
For description of the concurrency policies, see Parallel Objects.

retry_policy
For a description of the retry policy, see IIOP Client Failover.

Sample ICF File

Listing 2-1 shows a sample ICF file.

Listing 2-1   Sample ICF

module POA_University1
{
  implementation CourseSynopsisEnumerator_i
  {
    activation_policy ( process );
    transaction_policy ( optional );
    implements ( University1::CourseSynopsisEnumerator );
  };

};

module POA_University1
{
  implementation Registrar_i
  {
    activation_policy ( method );
    transaction_policy ( optional );
    implements ( University1::Registrar );
  };

};
module POA_University1
{
  implementation RegistrarFactory_i
  {
    activation_policy ( process );
    transaction_policy ( optional );
    implements ( University1::RegistrarFactory );
  }
};

Creating the ICF File

You have the option of either coding the ICF file manually or using the genicf command to
generate it from the OMG IDL file. For a description of the syntax and options for the genicf
command, see the BEA Tuxedo Command Reference.

See Also

Steps for Creating a BEA Tuxedo CORBA Server Application in Creating CORBA Server
Applications
This topic includes the following sections:

- **A Simple Programming Model.** This section describes:
  - Control Flow
  - Object State Management
  - Transaction Integration
  - Object Housekeeping
  - High-level Services

- **State Management.** This section describes:
  - Activation Policy
  - Application-controlled Activation and Deactivation
  - Servant Lifetime
  - Saving and Restoring Object State

- **Transactions.** This section describes:
  - Transaction Policies
  - Transaction Initiation
  - Transaction Termination
  - Transaction Suspend and Resume
The BEA Tuxedo CORBA TP Framework provides a programming TP Framework that enables users to create servers for high-performance TP applications. This chapter describes the TP Framework programming model and the TP Framework application programming interface (API) in detail. Additional information about how to use this API can be found in Creating CORBA Server Applications.

The TP Framework is required when developing BEA Tuxedo CORBA servers. Later releases will relax this requirement, though it is expected that most customers will use the TP Framework as an integral part of their applications.

BEA Tuxedo provides the infrastructure for providing load balancing, transactional capabilities, and administrative infrastructure. The base API used by the TP Framework is the CORBA API with BEA extensions. The TP Framework API is exposed to customers. The BEA Tuxedo ATMI is an optional API that can be mixed in with TP Framework APIs, allowing a customer to deploy distributed applications using a mix of CORBA servers and ATMI servers.

Before BEA Tuxedo CORBA, ORB products did not approach BEA Tuxedo’s performance in large-scale environments. BEA Tuxedo systems support applications that can process hundreds of transactions per second. These applications are built using the BEA Tuxedo stateless-service programming model that minimizes the amount of system resources used for each request, and thus maximizes throughput and price performance.

Now, BEA Tuxedo CORBA and its TP Framework give customers a way to develop CORBA applications with performance similar to BEA Tuxedo ATMI applications. BEA Tuxedo CORBA servers provide throughput, response time, and price performance approaching the BEA Tuxedo stateless-service programming model, while using the CORBA programming model.
A Simple Programming Model

The TP Framework provides a simple, useful subset of the wide range of possible CORBA object implementation choices. You use it for the development of server-side object implementations only. When using any client-side CORBA ORB, clients interact with CORBA objects whose server-side implementations are managed by the TP Framework. Clients are unaware of the existence of the TP Framework—a client written to access a CORBA object executing in a non-BEA Tuxedo server environment will be able to access that same CORBA object executing in a BEA Tuxedo server environment without any changes or restrictions to the client interface.

The TP Framework provides a server environment and an API that is easier to use and understand than the CORBA Portable Object Adapter (POA) API, and is specifically geared towards enterprise applications. It is a simple server programming model and an orthodox implementation of the CORBA model, which will be familiar to programmers using ORBs such as ORBIX or VisiBroker.

The TP Framework simplifies the programming of BEA Tuxedo CORBA servers by reducing the complexity of the server environment in the following ways:

- The TP Framework does all interactions with the POA and the Naming Service. The application programmer requires no knowledge of POA or Naming Service interfaces.
- The TP Framework is single threaded—only one request on one CORBA object will be processed at a time, obviating the need to write thread-safe implementations.
- A CORBA object may be involved in only one transaction at a time (consistent with the association of one object ID to one servant).

The TP Framework provides the following functionality:

- Control Flow
- Object State Management
- Transaction Integration
- Object Housekeeping
- High-level Services

Control Flow

The TP Framework, in conjunction with the ORB and the POA, controls the flow of the application program by doing the following:
• Controlling the server mainline and invoking callback methods on TP Framework-defined classes at appropriate times for server startup and shutdown. This relieves the application programmer from complex interactions related to ORB and POA initialization and coordination of transactions, resource managers, and object state on shutdown.

• Scheduling objects for activation and deactivation when client requests arrive and are completed. This removes the complexity of management of object activation and deactivation from the realm of the application programmer and enables the use of the TP monitor infrastructure’s powerful load-balancing capabilities, crucial to performance of mission-critical tasks.

Object State Management

The TP Framework API provides callback methods for application code to implement flexible state management schemes for CORBA objects. State management involves the saving and restoring of object state on object deactivation and activation. It also concerns the duration of activation of objects, which influences the performance of servers and their resource usage. The default duration of object activation is controlled by policies assigned to implementations at IDL compile time.

Transaction Integration

TP Framework transaction integration provides the following features:

• CORBA objects can participate in global transactions.

• Objects participating in transactions can be implemented as stateful objects that remain in memory for the duration of a transaction (by using the transaction activation policy), to decrease client response time.

• CORBA objects that participate in transactions can affect transaction outcome either during their transactional work or just prior to the system’s execution of the two-phase commit algorithm after all transactional work has been completed.

• Transactions can be automatically initiated on the server transparent to the client.

Object Housekeeping

When a server is shut down, the TP Framework rolls back any transactions that the server is involved in and deactivates any CORBA objects that are currently active.
High-level Services

The TP interface in the TP Framework API provides methods for performing object registrations and utility functions. The following services are provided:

- Object reference creation
- Factory-based routing support
- Accessors for system objects, such as the ORB
- Registration and unregistration of factories with the FactoryFinder
- Application-controlled activation and deactivation
- User logging

The purpose of these high-level service methods is to eliminate the need for developers to understand the CORBA POA, CORBA Naming Service, and BEA Tuxedo APIs, which they use for their underlying implementations. By encapsulating the underlying API calls with a high-level set of methods, programmers can focus their efforts on providing business logic rather than understanding and using the more complex underlying facilities.

State Management

State management involves the saving and restoring of object state on object deactivation and activation. It also concerns the duration of activation of objects, which influences the performance of servers and their resource usage. The external API of the TP Framework provides `activate_object` and `deactivate_object` methods, which are a possible location for state management code.

Activation Policy

State management is provided in the TP Framework by the activation policy. This policy controls the activation and deactivation of servants for a particular IDL interface (as opposed to the creation and destruction of the servants). This policy is applicable only to CORBA objects using the TP Framework.

The activation policy determines the default in-memory activation duration for a CORBA object. A CORBA object is active in a POA if the POA's active object map contains an entry that associates an object ID with an existing servant. Object deactivation removes the association of
an object ID with its active servant. You can choose from one of three activation policies:

- **method** (the default), **transaction**, or **process**.

**Note:** The activation policies are set in an ICF file that is configured at OMG IDL compile time.

For a description of the ICF file, refer to the Implementation Configuration File (ICF) section.

The activation policies are described below:

- **method** (This is the default activation policy.)

  The activation of the CORBA object (that is, the association between the object ID and the servant) lasts until the end of the method. At the completion of a method, the object is deactivated. When the next method is invoked on the object reference, the CORBA object is activated (the object ID is associated with a new servant). This behavior is similar to that of a BEA Tuxedo stateless service.

- **transaction**

  The activation of the CORBA object (that is, the association between the object ID and the servant) lasts until the end of the transaction. During the transaction, multiple object methods can be invoked. The object is activated before the first method invocation on the object and is deactivated in one of the following ways:

  - If a user-initiated transaction is in effect when the object is activated, the object is deactivated when the first of the following occurs: the transaction is committed or rolled back, or the server is shut down in an orderly fashion. The latter is done using either the `tmshutdown` or `tmadmin` command. These commands are described in the BEA Tuxedo Command Reference online document.

  - If a user-initiated transaction is not in effect when the TP object is activated, the TP object is deactivated when the method completes.

The **transaction** activation policy provides a means for an object to vote on the outcome of the transaction prior to the execution of the two-phase commit algorithm. An object votes to roll back the transaction by calling `Current.rollback_only()` in the `Tobj_ServantBase::deactivate_object` method. It votes to commit the transaction by not calling `Current.rollback_only()` in the method.

**Note:** This is a model of resource allocation that is similar to that of a BEA Tuxedo conversational service. However, this model is less expensive than the BEA Tuxedo conversational service in that it uses fewer system resources. This is because of the BEA Tuxedo ORB’s multicontexted dispatching model (that is, the presence of many servants in memory at the same time for one server), which makes it possible for a single server process to be shared by many concurrently active servants that service
many clients. In the BEA Tuxedo system, the process would be dedicated to a single client and to only one service for the duration of a conversation.

- **process**

The activation of the CORBA object begins when it is invoked while in an inactive state and, by default, lasts until the end of the process.

**Note:** The TP Framework API provides an interface method (TP::deactivateEnable) that allows the application to control the timing of object deactivation for objects that have the activation policy set to process. For a description of this method, see the section TP::deactivateEnable().

## Application-controlled Activation and Deactivation

Ordinarily, activation and deactivation decisions are made by the TP Framework, as discussed earlier in this chapter. The techniques in this section show how to use alternate mechanisms. The application can control the timing of activation and deactivation explicitly for objects with particular policies.

### Explicit Activation

Application code can bypass the on-demand activation feature of the TP Framework for objects that use the process activation policy. The application can “preactivate” an object (that is, activate it before any invocation) using the TP::create_active_object_reference call.

Preactivation works as follows. Before the application creates an object reference, the application instantiates a servant and initializes that servant’s state. The application uses TP::create_active_object_reference to put the object into the Active Object Map (that is, associate the servant with an ObjectId). Then, when the first invocation is made, the TP Framework immediately directs the request to the process that created the object reference and then to the existing servant, bypassing the necessity to call Server::create_servant and then the servant’s activate_object method (just as if this were the second or later invocation on the object). Note that the object reference for such an object will not be directed to another server and the object will never go through on-demand activation as long as the object remains activated.

Since the preactivated object has the process activation policy, it will remain active until one of two events occurs: (1) the ending of the process or (2) a TP::deactivateEnable call.

**Usage Notes**

Preactivation is especially useful if the application needs to establish the servant with an initial state in the same process, perhaps using shared memory to initialize state. Waiting to initialize
state until a later time and in a potentially different process may be very difficult if that state includes pointers, object references, or complex data structures. 

TP::create_active_object_reference guarantees that the preactivated object is in the same process as the code that is doing the preactivation. While this is convenient, preactivation should be used sparingly, as should all process objects, because it preallocates precious resources. However, when needed and used properly, preallocation is more efficient than alternatives.

Examples of such usage might be an object using the “iterator” pattern. For example, there might a potentially long list of items that could be returned (in an unbound IDL sequence) from a “database_query” method (for example, the contents of the telephone book). Returning all such items in the sequence is impractical because the message size and the memory requirements would be too large.

On an initial call to get the list, an object using the iterator pattern returns only a limited number of items in the sequence and also returns a reference to an “iterator” object that can be invoked to receive further elements. This iterator object is initialized by the initial object; that is, the initial object creates a servant and sets its state to keep track of where in the long list of items the iteration currently stands (the pointer to the database, the query parameters, the cursor, and so forth).

The initial object preactivates this iterator object by using TP::create_active_object_reference. It also creates an object reference to that object to return to the client. The client then invokes repeatedly on the iterator object to receive, say, the next 100 items in the list each time. The advantage of preactivation in this situation is that the state might be complex. It is often easiest to set such state initially, from a method that has all the information in its context (call frame), when the initial object still has control.

When the client is finished with the iterator object, it invokes a final method on the initial object which deactivates the iterator object. The initial object deactivates the iterator object by invoking a method on the iterator object that calls the TP::deactivateEnable method, that is, the iterator object calls TP::deactivateEnable on itself.

**Caution to Users**

For objects to be preactivated in this fashion, the state usually cannot be recovered if a crash occurs. (This is because the state was considered too complex or inconvenient to set upon initial, delayed activation.) This is a valid object technique, essentially stating that the object is valid only for a single activation period.

However, a problem may arise because of the “one-time” usage. Since a client still holds an object reference that leads to the process containing that state, and since the state cannot be
recreated after the crash, care must be taken that the client’s next invocation does not automatically provoke a new activation of the object, because that object would have inapplicable state.

The solution is to refuse to allow the object to be activated automatically by the TP Framework. If the user provides the TobjS::ActivateObjectFailed exception to the TP Framework as a result of the activate_object call, the TP Framework will not complete the activation and will return an exception to the client, CORBA::OBJECT_NOT_EXIST. The client has presumably been warned about this possibility, since it knows about the iterator (or similar) pattern. The client must be prepared to restart the iteration.

**Note:** This defensive measure may not be necessary in the future; the TP Framework itself may detect that the object reference is no longer valid. In particular, you should not depend on the possibility that the activate_object method might be called. If the TP Framework does in fact change, activate_object will not be called and the framework itself will generate the OBJECT_NOT_EXIST exception.

**Self Deactivation**

Just as it is possible to preactivate an object with the process activation policy, it is possible to request the deactivation of an object with the process activation policy. The ability to preactivate and the ability to request deactivation are independent; regardless of how an object was activated, it can be deactivated explicitly.

A method in the application can request (via TP::deactivateEnable) that the object be deactivated. When TP::deactivateEnable is called and the object is subsequently deactivated, no guarantee is made that subsequent invocations on the CORBA object will result in reactivation in the same process as a previous activation. The association between the ObjectId and the servant exists from the activation of the CORBA object until one of the following events occurs: (1) the shutdown of the server process or (2) the application calls TP::deactivateEnable. After the association is broken, when the object is invoked again, it can be reactivated anywhere that is allowed by the BEA Tuxedo configuration parameters.

There are two forms of TP::deactivateEnable. In the first form (with no parameters), the object currently executing will be deactivated after completion of the method in which the call is made. The object itself makes the decision that it should be deactivated. This is often done during a method call that acts as a "signoff" signal.

The second form of TP::deactivateEnable allows a server to request deactivation of any active object, whether it is the object that is executing or not; that is, any part of the server can ask that the object be deactivated. This form takes parameters identifying the object to be
deactivated. Explicit deactivation is not allowed for objects with an activation policy of transaction, because such objects cannot be safely deactivated until the end of a transaction.

In the TP::deactivateEnable call, the TP Framework calls the servant’s deactivate_object method. Exactly when the TP Framework invokes deactivate_object depends on the state of the object to be deactivated. If the object is not currently in execution, the TP Framework deactivates it before returning to the caller. The object might be currently executing a method; this is always the case for TP::deactivateEnable with no parameters (since it refers to the currently executing object). In this case, TP::deactivateEnable is not told whether the object was deactivated immediately or not.

Note: The TP::deactivateEnable(interface, object id, servant) method can be used to deactivate an object. However, if that object is currently in a transaction, the object will be deactivated when the transaction commits or rolls back. If an invoke occurs on the object before the transaction is committed or rolled back, the object will not be deactivated.

To ensure the desired behavior, make sure that the object is not in a transaction or ensure that no invokes occur on the object after the TP::deactivateEnable() call until the transaction is complete.

**Servant Lifetime**

A servant is a C++ class that contains methods to implement an IDL interface’s operations. The user writes the servant code. The TP Framework invokes methods in the servant code to satisfy requests. The servant is created by the C++ “new” statement and is destroyed by the C++ “delete” statement. Exactly who does the creation and who does the deletion, and the timing of creation and deletion, is the subject of this section.

**The Normal Case**

In the normal case, the TP Framework completely controls the lifetime of a servant. The basic model is that, when a request for an inactive object arrives, the TP Framework obtains a servant and then activates it (by calling its activate_object method). At deactivation time, the TP Framework calls the servant’s deactivate_object method and then disposes of the servant.

The phase “the TP Framework obtains a servant” means that when the TP Framework needs a servant to be created, it calls a user-written Server method, either Server::create_servant or ServerBase::create_servant_with_id. At that time, the application code must return a pointer to the requested servant. The application almost always does this by using the C++ “new”
statement to create a new instance of a servant. The phrase “disposes of the servant” means that
the TP Framework removes the reference to the servant, which actually deletes it.

The application must be aware that this current behavior of always creating and removing a
servant may change in future versions of this product. The application should not depend on the
current behavior, but should write servant code that allows reuse of a servant. Specifically, the
servant code must work even if the servant has not been freshly created (by the C++ “new”
statement). The TP Framework reserves the right not to remove a servant after it has been
deactivated and then to reactivate it. This means that the servant must completely initialize itself
at the time of the callback on the servant’s activate_object method, not at the time of servant
creation (not in the constructor).

Special Cases
There are two techniques an application can use to alter the normal TP Framework use of
servants. The first has to do with obtaining a servant and the second has to do with disposing of
the servant.

The application can alter the “obtaining” mechanism by using explicit preactivation. In this case,
the application creates and initializes a servant before asking the TP Framework to declare it
activated. Once such a servant has been turned over to the TP Framework (by the
TP::create_active_object_reference call), that servant is treated by the TP Framework
just like every other servant. The only difference is in its method of creation and initialization.

The application can alter the “disposing” mechanism by taking the responsibility for disposing of
a servant instead of leaving that responsibility with the TP Framework. Once a servant is known
to the TP Framework (through Server::create_servant,
ServerBase::create_servant_with_id, or TP::create_active_object_reference),
the TP Framework’s default behavior is to remove that servant itself. In this case, the application
code must no longer use references to the servant after deactivation.

However, the application may tell the TP Framework not to dispose of the servant after the TP
Framework deactivates it. Taking responsibility for a servant is done on an individual servant
basis, not for a whole class of servants, by calling Tobj_ServantBase::_add_ref with a
parameter identifying the servant.

Note: In applications written using BEA Tuxedo release 8.0 or later, use the
Tobj_ServantBase::_add_ref method instead of the
TP::application_responsibility() method. Unlike the
TP::application_responsibility() method, the add_ref() method takes no
arguments.
The advantage of the application taking responsibility for the servant is that the servant does not have to be created anew. If obtaining the servant is an expensive proposition, the application may choose to save the servant and reuse it later. This is especially likely to be true for servants for preactivated objects, but is true in general. For example, the next time the TP Framework makes a call on Server::create_servant or ServerBase::create_servant_with_id, the application might return a previously saved servant.

Additionally, once an application has taken responsibility for a servant, the application must take care to remove the servant (using Tobj_ServantBase::_remove_ref) when the servant is no longer needed, that is, when the reference count drops to zero, the same as for any other C++ instance. For more information about how the _remove_ref() method works, see Tobj_ServantBase::_remove_ref().

For more information on writing single-threaded and multithreaded server applications, see Creating CORBA Server Applications.

Saving and Restoring Object State

While CORBA objects are active, their state is contained in a servant. Unless an application uses TP::create_active_object_reference, state must be initialized when the object is first invoked (that is, the first time a method is invoked on a CORBA object after its object reference is created), and on subsequent invocations after they have been deactivated. While a CORBA object is deactivated, its state must be saved outside the process in which the servant was active. The object’s state can be saved in shared memory, in a file, or in a database. Before a CORBA object is deactivated, its state must be saved, and when it is activated, its state must be restored.

The programmer determines what constitutes an object’s state and what must be saved before an object is deactivated, and restored when an object is activated.

Note On Use of Constructors and Destructors for CORBA Objects

The state of CORBA objects must not be initialized, saved, or restored in the constructors or destructors for the servant classes. This is because the TP Framework may reuse an instance of a servant rather than deleting it at deactivation. No guarantee is made as to the timing of the creation and deletion of servant instances.

Transactions

The following sections provide information about transaction policies and how to use transactions.
Transaction Policies

Eligibility of CORBA objects to participate in global transactions is controlled by the transaction policies assigned to implementations at compile time. The following policies can be assigned.

Note: The transaction policies are set in an ICF file that is configured at OMG IDL compile time. For a description of the ICF file, refer to the Implementation Configuration File (ICF) section.

- **never**
  
  The implementation is not transactional. Objects created for this interface can never be involved in a transaction. The system generates an exception (INVALID_TRANSACTION) if an implementation with this policy is involved in a transaction. An AUTOTRAN policy specified in the UBBCONFIG file for the interface is ignored.

- **ignore**
  
  The implementation is not transactional. This policy instructs the system to allow requests within a transaction to be made of this implementation. An AUTOTRAN policy specified in the UBBCONFIG file for the interface is ignored.

- **optional** (This is the default transaction policy.)
  
  The implementation may be transactional. Objects can be involved in a transaction if the request is transactional. Servers containing transactional objects must be configured within a group associated with an XA-compliant resource manager. If the AUTOTRAN parameter is specified in the UBBCONFIG file for the interface, AUTOTRAN is on.

- **always**
  
  The implementation is transactional. Objects are required to always be involved in a transaction. If a request is made outside a transaction, the system automatically starts a transaction before invoking the method. The transaction is committed when the method ends. (This is the same behavior that results from specifying AUTOTRAN for an object with the option transaction policy, except that no administrative configuration is necessary to achieve this behavior, and it cannot be overridden by administrative configuration.) Servers containing transactional objects must be configured within a group that is associated with an XA-compliant resource manager.

Note: The optional policy is the only transaction policy that can be influenced by administrative configuration. If the system administrator sets the AUTOTRAN attribute for the interface by means of the UBBCONFIG file or by using administrative tools, the system automatically starts a transaction upon invocation of the object, if it is not already infected with a transaction (that is, the behavior is as if the always policy were specified).
Transaction Initiation

Transactions are initiated in one of two ways:

- By the application code via use of the `CosTransactions::Current::begin()` operation. This can be done in either the client or the server. For a description of this operation, see *Using CORBA Transactions*.

- By the system when an invocation is done on an object which has either:
  - Transaction policy `always`
  - Transaction policy `optional` and a setting of `AUTOTRAN` for the interface

For more information, see *Using CORBA Transactions*.

Transaction Termination

In general, the handling of the outcome of a transaction is the responsibility of the initiator. Therefore, the following are true:

- If the client or server application code initiates transactions, the TP Framework never commits a transaction. The BEA Tuxedo system may roll back the transaction if server processing tries to return to the client while the transaction is in an illegal state.

- If the system initiates a transaction, the commit or rollback will always be handled by the BEA Tuxedo system.

The following behavior is enforced by the BEA Tuxedo system:

- If no transaction is active when a method on a CORBA object is invoked and that method begins a transaction, the transaction must be either committed, rolled back, or suspended when the method invocation returns. If none of these actions is taken, the transaction is rolled back by the TP Framework, and the `CORBA::OBJ_ADAPTER` exception is raised to the client application. This exception is raised because the transaction was initiated in the server application; therefore, the client application would not expect a transactional error condition such as `TRANSACTION_ROLLEDBACK`.

Transaction Suspend and Resume

The CORBA object must follow strict rules with respect to suspending and resuming a transaction within a method invocation. These rules and the error conditions that result from their violation are described below.
When a CORBA object method begins execution, it can be in one of the following three states with respect to transactions:

- The client application began the transaction.
  - Legal server application behavior: Suspend and resume the transaction within the method execution.
  - Illegal server application behavior: Return from the method with the transaction in the suspended state (that is, return from the method without invoking resume if suspend was invoked).
  - Error Processing: If illegal behavior occurs, the TP Framework raises the CORBA::TRANSACTION_ROLLEDBACK exception to the client application and the transaction is rolled back by the BEA Tuxedo system.

- The system began a transaction to provide AUTOTRAN or transaction policy always behavior.
  - Legal server behavior: Suspend and resume the transaction within the method execution.
  - Note: Not recommended. The transaction may be timed out and aborted before the method causes the transaction to be resumed.
  - Illegal server behavior: Return from the method with the transaction in the suspended state (that is, return from the method without invoking resume if suspend was invoked).
  - Error Processing: If illegal behavior occurs, the TP Framework raises the CORBA::OBJ_ADAPTER exception to the client, and the transaction is rolled back by the system. The CORBA::OBJ_ADAPTER exception is raised because the client application did not initiate the transaction, and, therefore, does not expect transaction error conditions to be raised.

- The CORBA object is not involved in a transaction when it starts executing.
  - Legal server behavior:
    - Begin and commit a transaction within the method execution.
    - Begin and roll back a transaction within the method execution.
    - Begin and suspend a transaction within the method execution.
– **Illegal server behavior:** Begin a transaction and return from the method with the transaction active.

– **Error Processing:** If illegal behavior occurs, the TP Framework raises the `CORBA::OBJ_ADAPTER` exception to the client application and the transaction is rolled back by the BEA Tuxedo system. The `CORBA::OBJ_ADAPTER` exception is raised because the client application did not initiate the transaction, and, therefore, does not expect transaction error conditions to be raised.

### Restrictions on Transactions

The following restrictions apply to BEA Tuxedo CORBA transactions:

- A CORBA object in the BEA Tuxedo system must have the same transaction context when it returns from a method invocation that it had when the method was invoked.

- A CORBA object can be infected by only one transaction at a time. If an invocation tries to infect an already infected object, a `CORBA::INVALID_TRANSACTION` exception is returned.

- If a CORBA object is infected with a transaction and a nontransactional request is made on it, a `CORBA::OBJ_ADAPTER` exception is raised.

- If the application begins a transaction in `Server::initialize()`, it must either commit or roll back the transaction before returning from the method. If the application does not, the TP Framework shuts down the server. This is because the application has no predictable way of regaining control after completing the `Server::initialize` method.

- If a CORBA object is infected by a transaction and with an activation policy of transaction, and if the reason code passed to the method is either `DR_TRANS_COMMITTING` or `DR_TRANS_ABORTED`, no invocation on any CORBA object can be done from within the `Tobj_ServantBase::deactivate_object` method. Such an invocation results in a `CORBA::BAD_INV_ORDER` exception.

### SQL and Global Transactions

Adhere to the following guidelines when using SQL and Global Transactions:

- Care should be taken when executing SQL statements outside the scope of a global transaction. The SQL standard specifies that a local transaction should be started implicitly by the database resource manager whenever an SQL statement that needs the context of a transaction is executed and no transaction is active. The standard also says that a transaction that is implicitly started by the database resource manager must then be
explicitly terminated by executing a COMMIT or ROLLBACK SQL statement; the TP Framework is not responsible for terminating transactions that are started by the resource manager.

**Note:** This is not an issue when an application uses the XA library to connect to the Oracle server because those applications can operate only on global transactions. The Oracle server does not allow local transactions when it is using XA.

- The SQL COMMIT and ROLLBACK statements cannot be used to terminate a global transaction that has been either started explicitly using `Current.begin()` or started implicitly by the system. Check the database vendor documentation for each database product for other possible restrictions when using global transactions.
- SQL cursors may be closed when transactions are terminated. Consult your database product documentation for exact information about cursor handling rules. Application programmers should be careful to use cursors only with CORBA objects with appropriate activation policies and within appropriate transaction boundaries.

### Voting on Transaction Outcome

CORBA objects can affect transaction outcome during two stages of transaction processing:

- **During transactional work**
  
  The `Current.rollback_only` method can be used to ensure that the only possible outcome is to roll back the current transaction. `Current.rollback_only()` can be invoked from any CORBA object method.

- **After completion of transactional work**

  CORBA objects that have the transaction activation policy are given a chance to vote whether the transaction should commit or roll back after transactional work is completed. These objects are notified of the completion of transactional work prior to the start of the two-phase commit algorithm when the TP Framework invokes their `deactivate_object` method.

  Note that this behavior does not apply to objects with `process` or `method` activation policies. If the CORBA object wants to roll back the transaction, it can call `Current::rollback_only`. If it wants to vote to commit the transaction, it does not make that call. Note, however, that a vote to commit does not guarantee that the transaction is committed, since other objects may subsequently vote to roll back the transaction.

  **Note:** Users of SQL cursors must be careful when using an object with the `method` or `process` activation policy. A process opens an SQL cursor within a client-initiated transaction. For typical SQL database products, once the client commits the transaction, all cursors...
that were opened within that transaction are automatically closed; however, the object will not receive any notification that its cursor has been closed.

Transaction Timeouts

When a transaction timeout occurs, the transaction is marked so that the only possible outcome is to roll back the transaction, and the CORBA::TRANSACTION_ROLLEDBACK standard exception is returned to the client. Any attempts to send new requests will also fail with the CORBA::TRANSACTION_ROLLEDBACK exception until the transaction has been aborted.

IIOP Client Failover

It is not always possible to determine when a server instance failed with respect to the work it was doing at the time of failure. For example, if a server instance fails after handling a client request but before returning the response, there is no way to tell that the request was handled. A user that does not get a response will most likely retry, resulting in an additional request.

Support for IIOP client failover has been added to BEA Tuxedo CORBA as an availability enhancement. IIOP client failover provides a transparent mechanism for a CORBA remote client to automatically connect to an alternative ISL and then retry the request in case of failure. IIOP client failover marks an interface implementation as idempotent. An idempotent implementation is one that can be repeated without any negative side-effects. For example, SET BALANCE.

Setting The Retry Policy

In order to mark an interface implementation as idempotent, you must set the retry policy in the implementation configuration file (ICF) using the retry_policy option. For a description of the ICF, refer to the Implementation Configuration File (ICF) section.

The retry_policy option has two settings:

- **never**: The default setting. It indicates that the interface implementation is not idempotent and that requests should never be automatically retried.

- **always**: Indicates that the interface implementation is idempotent and that requests should always be retried in case of failure.
MIB Support

You can also check the retry policy using the TA_RTPOLICY attribute added to the MIB(5) T_IFQUEUE and T_INTERFACE classes. The TA_RTPOLICY attribute value is either never or always.

Initiating IIOP Client Failover

To initiate IIOP client failover support, ISL servers must be specified using the -C warn|none option in the *SERVERS section of the UBBCONFIG file

This option allows ISL to accept unofficial connection directly from the client orb. ISL servers that are not specified using the -C warn|none option will not be placed in candidate IIOP gateway pools. Consequently, the client will not failover to those ISL servers.

In the following UBBCONFIG file example, the ISL servers specified in lines 1 and 2 will support client failover. The ISL server in line 3 will not.

Listing 3-1  Example UBBCONFIG File IIOP Client Failover Entry

```
*SERVERS
     ISL SRVGRP=SYS_GRP1 SRVID=10 CLOPT="-A -- -C warn -n //myhost1:2468"
     ISL SRVGRP=SYS_GRP2 SRVID=20 CLOPT="-A -- -C none -n //myhost2:2469"
     ISL SRVGRP=SYS_GRP3 SRVID=30 CLOPT="-A -- -n //myhost3:2470"
```

IIOP Client Failover Limitations

IIOP Client Failover is not supported under the following three instances:

- Tuxedo system-supplied object failover
  Only application-supplied object failover is supported.

- Transaction mode

- SSL link or authentication is required
See Also

- Steps for Creating a BEA Tuxedo CORBA Server Application in Creating CORBA Server Applications
- UBBCONFIG(5)
- MIB(5)
- T_IFQUEUE Class
- T_INTERFACE Class

Weblogic CORBA Clustering and Load Balancing Support

Tuxedo CORBA C++ client supports failover to Weblogic clustering servers and also supports load balancing. For more information, see WebLogic 9.1 documentation - Failover and Replication in a Cluster and Load Balancing in a Cluster.

Parallel Objects

Support for parallel objects was added to BEA Tuxedo CORBA in release 8.0 as a performance enhancement. The parallel objects feature enables you to designate all business objects in a particular application as stateless objects. The effect is that, unlike stateful business objects, which can only run on one server in a single domain, stateless business objects can run on all servers in a single domain. Thus, the benefits of parallel objects are as follows:

- Parallel objects can run on multiple servers in the same domain at the same time. Thus, utilization of all servers to service concurrent multiple requests improves performance.
- When the BEA Tuxedo system services requests to parallel business objects, it always looks for an available server to the local machine first. If all servers on the local machine are busy processing the requested business object, the BEA Tuxedo system looks for an available server on other machines in the local domain. Thus, if there are multiple servers on the local machine, network traffic is reduced and performance is improved.

For more information on parallel objects, see Scaling, Distributing, and Tuning CORBA Applications.

To implement parallel objects, the concurrency policy option has been added to the ICF file. To select parallel objects for a particular application, you set the concurrency policy option to user-controlled. When you select user-controlled concurrency, the business object are not
registered with the Active Object Map (AOM) and, therefore, are stateless and can be active on more than one server at a time. Thus, these objects are referred to as parallel objects.

If user-controlled concurrency is selected, the servant implementation must comply with one of the following statements:

- The servant implementation must have no requirements for concurrent access to a shared resource
- Or the servant implementation must utilize some other tool (for example, a database and locking) to ensure the correct behavior during concurrent access to resources.

In release 8.0 of the BEA Tuxedo software, the Implementation Configuration File (ICF) was modified to support user-controlled concurrency. In Listing 3-2, the changes to add this support are highlighted in **bold** type. For a description of the ICF syntax, see ICF Syntax.

**Listing 3-2**  ICF Syntax

```
[#pragma activation_policy method|transaction|process]
[#pragma transaction_policy never|ignore|optional|always]
[#pragma concurrency_policy user_controlled|system_controlled]
[Module module-name {
  implementation [implementation-name]
  
  implements (module-name::interface-name);
  [activation_policy (method|transaction|process);]
  [transaction_policy (never|ignore|optional|always);]
  [concurrency_policy (user_controlled|system_controlled);]
  ];
};
]
```

User-controlled concurrency can be used with factory-based routing, all activation policies, and all transaction policies. The interaction with these features is as follows:

- Factory-based routing

  If the user specifies factory-based routing when creating the object, then the object will route to a server in that group. The object key contains the group selected during factory-based routing, but the client routing code will recognize that the interface has user-controlled concurrency and specify the desired group. This is accomplished using normal BEA Tuxedo routing.
• Activation policy

The TP Framework handles active user-controlled concurrency objects in the same manner as system-controlled concurrency objects. The TP Framework stores information about objects in the local AOM, and calls the activate_object and deactivate_object methods at the appropriate times. However, the object will not have an entry in the AOM and the TP Framework will not call any AOM routines. For example, on shutdown, since an active object will not have an AOM handle, calls to remove the entry from the AOM will not be invoked.

• Transaction policy

The TP Framework handles active user-controlled concurrency objects in the same manner as system-controlled concurrency objects. The TP Framework is called back for transaction events and the TP Framework stores information about transactional user-controlled objects in the local AOM. The main differences when using parallel objects in transactions as opposed to stateful objects are that the AOM is not used for GTRID information and the AOM routines are not called to update or retrieve transactional information.

Note: There is one restriction with user-controlled concurrency. TP::create_active_object_reference throws a TobjS::IllegalOperation exception if it is passed an interface with user-controlled concurrency set. Since the AOM is not used when user-controlled concurrency is set, there is no way for the TP Framework to connect an active object to this server.

TP Framework API

This section describes the TP Framework API. Additional information about how to use this API can be found in Creating CORBA Server Applications.

The TP Framework comprises the following components:

• The Server C++ class, which has virtual methods for application-specific server initialization and termination logic

• The ServerBase C++ class, which has virtual methods for multithreaded server applications.

• The Tobj_ServantBase C++ class, which has virtual methods for object state management

• The TP C++ class, which provides methods to:
  – Create object references for CORBA objects

3-22   CORBA Programming Reference
The visible part of the TP Framework consists of two categories of operations:

- Service methods that can be called by user code. These are in the TP interface.
- Callback methods that are written by the user and that are invoked by the TP Framework. This includes methods in the Tobj_ServantBase and Server classes. These operations are intended to be called by TP Framework code only. The application code should never call the methods of these classes. If it does, unpredictable results may occur.

### Server Interface

The Server interface provides callback methods that can be used for application-specific server initialization and termination logic. This interface also provides a callback method that is used to create servants when servants are required for object activation.

The Server interface has the following characteristics:

- The Server class inherits from the ServerBase class.
- The Server class is a C++ native class.
- The Server.h file contains the declarations and definitions for the Server class.

For a description of the Server interface methods, see [ServerBase Interface](#).

### C++ Declarations

For the C++ mappings, see [ServerBase Interface](#).
ServerBase Interface

The serverBase interface allows you to take full advantage of multithreading capabilities. You can create your own Server classes that inherit from the ServerBase class. This provides you with the following:

- The create_servant_with_id() method to support implementations requiring knowledge of the target object during the creation of a servant
- Support for user-supplied thread initialization and release handlers

The ServerBase class provides the same operations that were available in the Server class in earlier releases. The Server class inherits from the ServerBase class.

These methods can be used with single-threaded and multithreaded applications:

- Server::create_servant()
- Server::initialize()
- Server::release()
- ServerBase::create_servant_with_id()

These methods can be used with multithreaded server applications only:

- ServerBase::thread_initialize()
- ServerBase::thread_release()

**Note:** Programmers must provide definitions of the Server class methods. The ServerBase class methods have default implementations.

C++ Declarations (in Server.h)

The C++ mapping is as follows:

```cpp
class OBEXPFDLLUSER ServerBase {
public:
    virtual CORBA::Boolean initialize(int argc, char** argv) = 0;
    virtual void release() = 0;
    virtual Tobj_Servant create_servant(const char* interfaceName) = 0;
};
```
// Default Implementations Supplied
virtual Tobj_Servant
    create_servant_with_id(const char* interfaceName,
                        const char* stroid);

virtual CORBA::Boolean
    thread_initialize(int argc, char** argv);

virtual void
    thread_release();
};

class Server : public ServerBase {
public:
    CORBA::Boolean initialize(int argc, char** argv);
    void release();
    Tobj_Servant create_servant(const char* interfaceName);
};

Server::create_servant()

Synopsis
Creates a servant to instantiate a C++ object.

C++ Binding
    class Server {
    public:
        Tobj_Servant create_servant(const char* interfaceName);
    };

Argument
    interfaceName
    Specifies a character string that contains the fully qualified interface name for the object. This will be the same interface name that was supplied when the object reference was created (TP::create_object_reference() or TP::create_active_object_reference()) for the object reference used for this invocation. This name can be used to determine which servant needs to be constructed.
Exception

If an exception is thrown in Server::create_servant(), the TP Framework catches the exception. Activation fails. A CORBA::OBJECT_NOT_EXIST() exception is raised back to the client. In addition, an error message is written to the user log (ULOG) file, as follows, for each exception type:

TobjS::CreateServantFailed
"TPFW_CAT:23: ERROR: Activating object - application raised
TobjS::CreateServantFailed. Reason = reason. Interface = interfaceName,
OID = oid"

Where reason is a user-supplied reason, and interfaceName and oid are the interface ID and object ID, respectively, of the invoked CORBA object.

TobjS::OutOfMemory
"TPFW_CAT:22: ERROR: Activating object - application raised
TobjS::OutOfMemory. Reason = reason. Interface = interfaceName, OID = oid"

Where reason is a user-supplied reason, and interfaceName and oid are the interface ID and object ID, respectively, of the invoked CORBA object.

CORBA::Exception
"TPFW_CAT:28: ERROR: Activating object - CORBA Exception not handled
by application. Exception ID = exceptionID. Interface = interfaceName,
OID = oid"

Where exceptionID is the interface ID of the exception, and interfaceName and oid are the interface ID and object ID, respectively, of the invoked CORBA object.

Other Exception
"TPFW_CAT:29: ERROR: Activating object - Unknown Exception not handled
by application. Exception ID = exceptionID. Interface = interfaceName,
OID = oid"

Where exceptionID is the interface ID of the exception, and interfaceName and oid are the interface ID and object ID, respectively, of the invoked CORBA object.

Description

The create_servant method is invoked by the TP Framework when a request arrives at the server and there is no available servant to satisfy the request. The TP Framework calls the create_servant method with the interface name for the servant to be created. The server application instantiates an appropriate C++ object and returns a pointer to it. Typically, the
method contains a switch statement on the interface name and creates a new object, depending on the interface name.

**Caution:** The server application must not depend on this method being invoked for every activation of a CORBA object. The server application must not do any handling of CORBA object state in the constructors or destructors of any servant classes for CORBA objects. This is because the TP Framework may possibly reuse servants on activation and may possibly not destroy servants on deactivation.

**Return Value**

Tobj_Servant

The pointer to the newly created servant (instance) for the specified interface. A NULL value should be returned if `create_servant()` is invoked with an interface name that it does not recognize or if the servant cannot be created for some reason.

If the `create_servant` method returns a NULL pointer, activation fails. A CORBA::OBJECT_NOT_EXIST() exception is raised back to the client. Also, the following message is written to the user log (ULOG):

"TPFW_CAT:23: ERROR: Activating object - application raised TobjS::CreateServantFailed. Reason = Application's Server::create_servant returned NULL. Interface = interfaceName, OID = oid"

Where `interfaceName` is the interface ID of the invoked interface and `oid` is the corresponding object ID.

**Note:** The restriction on the length of the ObjectId has been removed in this release.

**ServerBase::create_servant_with_id()**

**Synopsis**

Creates a servant for this target object. This method supports the development of single-headed and multithreaded server applications.

**C++ Binding**

```c++
Tobj_Servant create_servant_with_id (const char* interfaceName,
                                      const char* stroid);
```
Arguments

\textit{interfaceName}

Specifies a character string containing the fully qualified interface name for the object. This must be the same interface name that was supplied when the object reference was created.

\textit{stroid}

Specifies an object ID in string format. The object ID uniquely identifies the object associated with the request to be processed. This is the same object ID that was specified when the object reference was created.

Description

The TP Framework invokes the \texttt{create\_servant\_with\_id} method when a request arrives at the server and there is no servant available to satisfy the request. The TP Framework passes in the interface name for the servant to be created and the object ID associated with the object with which the servant will be associated. The server application instantiates an appropriate C++ object and returns a pointer to it. Typically, the method contains a \texttt{switch} statement on the interface name and creates a new object, depending on the interface name. Providing the object ID allows a servant implementation to make decisions during the creation of the servant instance that require knowledge of the target object. Reentrancy support is one example of how a servant implementation might employ knowledge of the target object.

The \texttt{ServerBase} class provides a default implementation of \texttt{create\_servant\_with\_id} which calls the standard \texttt{create\_servant} method passing the interface name. This default implementation ignores the target object ID parameter.

\textbf{Caution:} The server application must not depend on the invocation of this method for every activation of a CORBA object. The server application must not handle the CORBA object state in the constructors or destructors of any servant classes for CORBA objects. This is because the TP Framework might reuse servants on activation and might not destroy servants on deactivation.

Return Value

\texttt{Tobj\_Servant}

A pointer to the newly created servant (instance) for the specified interface. Returns \texttt{NULL} if either of these conditions is true:

- Interface name not recognized
- Unable to create a servant
Example

```cpp
Tobj_Servant simple_per_request_server::create_servant_with_id(
    const char* intf_repos_id, const char* stroid)
{
    TP::userlog("create_servant_with_id called in thread %ld",
                 (unsigned long)SIMPTHR_GETCURRENTTHREADID);

    // Perform any necessary initialization based on
    // this object ID

    return create_servant(intf_repos_id);
}
```

**Server::initialize()**

**Synopsis**

Allows the application to perform application-specific initialization procedures, such as logging
into a database, creating and registering well-known object factories, initializing global variables,
and so forth.

**C++ Binding**

```cpp
class Server {
public:
    CORBA::Boolean initialize(int argc, char** argv);
};
```

**Arguments**

The `argc` and `argv` arguments are passed from the command line. The `argc` argument contains
the name of the server. The `argv` argument contains the first command-line option that is specific
to the application, if there are any.

Command-line options are specified in the `UBBCONFIG` file using the `CLOPT` parameter in the
entry for the server in the `SERVERS` section. System-recognized options come first in the `CLOPT`
parameter, followed by a double-hyphen (`--`), followed by the application-specific options. The
value of `argc` is one greater than the number of application-specific options. For details, see `ubbconfig(5)` in the *File Formats, Data Descriptions, MIBs, and System Processes Reference*. 

TP Framework API
Exceptions

If an exception is raised in `Server::initialize()`, the TP Framework catches the exception. The TP Framework behavior is the same as if `initialize()` returned FALSE (that is, an exception is considered to be a failure). In addition, an error message is written to the user log (ULOG) file, as follows, for each exception type:

- `TobjS::InitializeFailed`
  ```
  Reason = reason"
  ```
  Where `reason` is a string supplied by application code. For example:
  ```
  Throw TobjS::InitializeFailed("Couldn't register factory");
  ```
- `CORBA::Exception`
  ```
  "TPFW_CAT:1: ERROR: Exception in Server::initialize(): exception.
  Reason = unknown"
  ```
  Where `exception` is the interface ID of the CORBA exception that was raised.
- Other Exceptions
  ```
  TPFW_CAT:1: ERROR: Exception in Server::initialize(): unknown exception. Reason = unknown"
  ```

Description

The `initialize` callback method, which is invoked as the last step in server initialization, allows the application to perform application-specific initialization.

Typically, a server application does the following tasks in `Server::initialize`:

- Creates references for CORBA object factories implemented in the server application and registers them with the FactoryFinder using the `TP::register_factory()` operation.
- Initializes global variables, if any are used.
- Opens XA resource managers if any are used by the server application.

It is the responsibility of the server application to open any required XA resource managers. This is done by invoking either of the following methods:

- `TP::open_xa_rm()`
  This is the preferred technique for server applications, since it can be done on a static function, without the need to obtain an object reference.
Note: You must use the `TP::open_xa_rm()` method if you use the INS bootstrap mechanism to obtain initial object references.

- `Tobj::TransactionCurrent::open_xa_rm()`
  A reference to the TransactionCurrent object can be obtained from the Bootstrap object. For an explanation of how to obtain a reference to the Bootstrap object, see the section `TP::bootstrap()`. For more information about the TransactionCurrent object, see the `CORBA Bootstrapping Programming Reference` section and Using CORBA Transactions.

- Transactions may be started in the `initialize` method after invoking the `Tobj::TransactionCurrent::open_xa_rm()` or `TP::open_xa_rm` method. However, any transactions that are started in `initialize()` must be terminated by the server application before `initialize()` returns. If the transactions are still active when control is returned, the server application fails to boot, and it exits gracefully. This happens because the server application has no logical way of either committing or rolling back the transaction after `Server::initialize()` returns. This condition is an error.

Return Value

Boolean `TRUE` or `FALSE`. `TRUE` indicates success. `FALSE` indicates failure. If an error occurs in `initialize()`, the application code should return `FALSE`. The application code should not call the system call `exit()`. Calling `exit()` does not give the TP Framework a chance to release resources allocated during startup and may cause unpredictable results.

If the return value is `FALSE`:

- `Server::release()` is not invoked.

- Any transactions that are started in the `initialize()` method and are not terminated will eventually time out; they are not automatically rolled back.

`ServerBase::thread_initialize()`

Synopsis

Performs any necessary application-specific initialization for a thread created using the BEA Tuxedo software. This method supports the development of a multithreaded server application.

C++ Binding

```cpp
CORBA::Boolean thread_initialize(int argc, char** argv)
```
Arguments

argc
The number of arguments provided to the application. Initially, this count is passed to the main function.

argv
The arguments provided to the application. Initially, these arguments are passed to the main function.

Description

In managing the thread pool, the BEA Tuxedo software creates and releases threads using the operating system thread library services. Depending on application requirements, these threads might need to be initialized before they are used to process requests.

The thread_initialize callback method is invoked each time a thread is created, to initialize the thread. Note that the BEA Tuxedo software manages a number of system-owned threads that are used for dispatching requests; these system-owned threads are in addition to those threads in the thread pool. Under some circumstances the servant methods you implement are also executed in these system-owned threads; for this reason the BEA Tuxedo software invokes the thread_initialize method to initialize the system-owned threads.

The ServerBase class provides a default implementation of the thread_initialize method that opens the XA resource manager in the initialized thread.

Return Value

CORBA::Boolean
True if the initialization of the thread was successful.

Example

CORBA::Boolean simple_per_request_server::thread_initialize(
    int argc, char** argv)
{
    TP::userlog("thread_initialize called in thread %ld",
        (unsigned long)SIMPTHR_GETCURRENTTHREADID);
    return CORBA_TRUE;
}
Server::release()

Synopsis

Allows the application to perform any application-specific cleanup, such as logging off a database, unregistering well-known factories, or deallocating resources.

C++ Binding

typedef Tobj_ServantBase* Tobj_Servant;

class Server {
    public:
        void release();
    };

Arguments

None.

Exceptions

If an exception is raised in release(), the TP Framework catches the exception. Each exception causes an error message to be written to the user log (ULOG) file, as follows:

TobjS::ReleaseFailed
  "TPFW_CAT:2: WARN: Exception in Server::release():
   IDL:beasys.com/TobjS/ReleaseFailed:1.0. Reason = reason"

Where reason is a string supplied by application code. For example:

  Throw TobjS::ReleaseFailed(
    "Couldn't unregister factory");

CORBA::Exception
  "TPFW_CAT:2: WARN: Exception in Server::release(): exception. Reason = unknown"

Where exception is the interface ID of the CORBA exception that was raised.

Other Exceptions

  "TPFW_CAT:2: WARN: Exception in Server::release(): unknown exception.
   Reason = unknown"

In all cases, the server continues to exit.
Description

The release callback method, which is invoked as the first step in server shutdown, allows the server application to perform any application-specific cleanup. The user must override the virtual function definition.

Typical tasks performed by the application in this method are as follows:

- Close XA resource managers.
- Unregister CORBA object factories that were registered with the FactoryFinder in Server::initialize().
- Deallocate any server resources not yet released.

This method is normally called in response to a tmshutdown command from the administrator or operator.

The TP Framework provides a default implementation of Server::release(). The default implementation closes XA resource managers for the server. The implementation does this by issuing a tx_close() invocation, which uses the default CLOSEINFO configured for the server's group in the UBBOUNDARY file.

It is the responsibility of the application to close any open XA resource managers. This is done by issuing either of the following calls:

- TP::close_xa_rm()

Note: You must use the TP::close_xa_rm() method if you use the INS bootstrap mechanism to obtain initial object references.

- Tobj::TransactionCurrent::close_xa_rm(). A reference to the TransactionCurrent object can be obtained from the Bootstrap object. For an explanation of how to obtain a reference to the Bootstrap object, see the section TP::bootstrap(). For more information about the TransactionCurrent object, see CORBA Bootstrapping Programming Reference and Using CORBA Transactions.

Note: Once a server receives a request from the tmshutdown(1) command to shut down, it can no longer receive requests from other remote objects. This may require servers to be shut down in a specific order. For example, if the Server::release() method in Server 1 needs to access a method of an object that resides in Server 2, Server 2 should be shut down after Server 1 is shut down. In particular, the TP::unregister_factory() method accesses the FactoryFinder Registrar object that resides in a separate server. The TP::unregister_factory() method is typically invoked from the release()
method; therefore, the FactoryFinder server should be shut down after all servers that call
TP::unregister_factory() in their Server::release() method.

Return Value
None.

ServerBase::thread_release()

Synopsis
Performs application-specific cleanup when a thread that was created by the BEA Tuxedo
software is released. This method supports the development of a multithreaded server
application.

C++ Binding
void thread_release()

Arguments
None.

Description
The thread_release callback method is invoked each time a thread is released. Implement the
thread_release method as necessary to perform application-specific resource cleanup.

The ServerBase class provides a default implementation of the thread_release method that
closes the XA resource manager in the released thread.

Return Value
None.

Example
void simple_per_request_server::thread_release()
{
    TP::userlog("thread_release called in thread %ld",
               (unsigned long)SIMPTHR_GETCURRENTTHREADID);
}
**Tobj_ServantBase Interface**

The `Tobj_ServantBase` interface inherits from the `PortableServer::RefCountServantBase` class and defines operations that allow a CORBA object to assist in the management of its state in a thread-safe manner. Every implementation skeleton generated by the IDL compiler automatically inherits from the `Tobj_ServantBase` class. The `Tobj_ServantBase` class contains two virtual methods, `activate_object()` and `deactivate_object()`, that may be optionally implemented by the programmer.

Whenever a request comes in for an inactive CORBA object, the object is activated and the `activate_object()` method is invoked on the servant. When the CORBA object is deactivated, the `deactivate_object()` method is invoked on the servant. The timing of deactivation is driven by the implementation’s activation policy. When the `deactivate_object()` method is invoked, the TP Framework passes in a reason code to indicate why the call was made.

These methods support the development of a multithreaded server application:

- `TobjServantBase::_add_ref()`
- `TobjServantBase::_is_reentrant()`
- `TobjServantBase::_remove_ref()`

**Note:** `Tobj_ServantBase::activate_object()` and `Tobj_ServantBase::deactivate_object()` are the only methods that the TP Framework guarantees will be invoked for CORBA object activation and deactivation. The servant class constructor and destructor may or may not be invoked at activation or deactivation time (through the `Server::create_servant` call for C++). Therefore, the server application code must not do any state handling for CORBA objects in either the constructor or destructor of the servant class.

**Note:** The programmer does not need to use a cast or reference to `Tobj_ServantBase` directly. The `Tobj_ServantBase` methods show up as part of the skeleton and, therefore, in the implementation class for a servant. The programmer may provide definitions for the `activate_object` and `deactivate_object` methods, but the programmer should never make direct invocations on those methods; only the TP Framework should call those methods.

**C++ Declaration (in Tobj_ServantBase.h)**

The C++ mapping for the `Tobj_servantBase` interface is as follows:

```cpp
class Tobj_ServantBase : public PortableServer::RefCountServantBase {
public:
```
Tobj_ServantBase& operator=(const Tobj_ServantBase&);
Tobj_ServantBase() {}
Tobj_ServantBase(const Tobj_ServantBase& s) :
   PortableServer::RefCountServantBase(s) {}

virtual void activate_object(const char *) {}
virtual void deactivate_object(const char*,
   TobjS::DeactivateReasonValue) {}

virtual CORBA::Boolean _is_reentrant() { return CORBA_FALSE; }
};
typedef Tobj_ServantBase * Tobj_Servant;

Tobj_ServantBase:: activate_object()

Synopsis
Associates an object ID with a servant. This method gives the application an opportunity to
restore the object’s state when the object is activated. The state may be restored from shared
memory, from an ordinary flat file, or from a database file.

C++ Binding
class Tobj_ServantBase : public PortableServer::ServantBase {
public:
   virtual void activate_object(const char * stroid) {}
};

Argument
stroid
Specifies the object ID in string format. The object ID uniquely identifies this instance of
the class. This is the same object ID that was specified when the object reference was
created (using TP:create_object_reference()) or in the
TP::create_active_object_reference() for the object reference used for this
invocation.

Note: The restriction on the length of the object ID has been removed in this release.

Description
Object activation is triggered by a client invoking a method on an inactive CORBA object. This
causes the Portable Object Adapter (POA) to assign a servant to the CORBA object. The
activate_object() method is invoked before the method invoked by the client is invoked. If
activate_object() returns successfully, that is, without raising an exception, the requested method is executed on the servant.

The activate_object() and deactivate_object() methods and the method invoked by the client can be used by the programmer to manage object state. The particular way these methods are used to manage object state may vary according to the needs of the application. For a discussion of how these methods might be used, see Creating CORBA Server Applications.

If the object is currently infected with a global transaction, activate_object() executes within the scope of that same global transaction.

It is the responsibility of the programmer of the object to check that the stored state of the object is consistent. In other words, it is up to the application code to save a persistent flag that indicates whether or not deactivate_object() successfully saved the state of the object. That flag should be checked in activate_object().

Return Value
None.

Exceptions
If an error occurs while executing activate_object(), the application code should raise either a CORBA standard exception or a TobjS::ActivateObjectFailed exception. When an exception is raised, the TP Framework catches the exception, and the following events occur:

- The activation fails.
- The method invoked by the client is not executed.
- If activate_object() is executing within a transaction and the client initiated the transaction, the transaction is not rolled back.
- A CORBA::OBJECT_NOT_EXIST exception is raised back to the client.

Note: For each CORBA interface, set AUTOTRAN to Yes if you want a transaction to start automatically when an operation invocation is received. Setting AUTOTRAN to Yes has no effect if the interface is already in transaction mode. For more information about AUTOTRAN, see Using CORBA Transactions.

- Based on the exception is raised, a message is written to the user log (ULOG) file, as follows:
TobjS::ActivateObjectFailed
"TPFW_CAT:24: ERROR: Activating object - application raised
TobjS::ActivateObjectFailed. Reason = reason. Interface =
interfaceName, OID = oid"

Where reason is a user-supplied reason, and interfaceName and oid are the interface ID and object ID, respectively, of the invoked CORBA object.

TobjS::OutOfMemory
"TPFW_CAT:22: ERROR: Activating object - application raised
TobjS::OutOfMemory. Reason = reason. Interface = interfaceName, OID = oid"

Where reason is a user-supplied reason, and interfaceName and oid are the interface ID and object ID, respectively, of the invoked CORBA object.

CORBA::Exception
"TPFW_CAT:25: ERROR: Activating object - CORBA Exception not handled
by application. Exception ID = exceptionID. Interface = interfaceName,
OID = oid"

Where exceptionID is the interface ID of the exception, and interfaceName and oid are the interface ID and object ID, respectively, of the invoked CORBA object.

Other exception
"TPFW_CAT:26: ERROR: Activating object - Unknown Exception not handled
by application. Exception ID = exceptionID. Interface = interfaceName,
OID = oid"

Where exceptionID is the interface ID of the exception, and interfaceName and oid are the interface ID and object ID, respectively, of the invoked CORBA object.

Tobj_ServantBase::_add_ref()

Synopsis

Adds a reference to a servant. This method supports the development of a multithreaded server application.

Note: In applications written using BEA Tuxedo release 8.0 or later, use this method instead of the TP::application_responsibility() method.

C++ Binding

void _add_ref()
Arguments
None.

Description
Invoke this method when a reference to a servant is needed. Invoking this method causes the
reference count for the servant to increment by one.

Return Value
None.

Example
myServant * servant = new intf_i();
if(servant != NULL)
  servant->_add_ref();

Tobj_ServantBase::deactivate_object()

Synopsis
Removes the association of an object ID with its servant. This method gives the application an
opportunity to save all or part of the object’s state before the object is deactivated. The state may
be saved in shared memory, in an ordinary flat file, or in a database file.

C++ Binding
class Tobj_ServantBase : public PortableServer::ServantBase {
public:
  virtual void deactivate_object(const char* stroid,
    TobjS::DeactivateReasonValue reason) {} 
};

Arguments
stroid
  Specifies the object ID in string format. The object ID uniquely identifies this instance of
  the class.

  Note: The restriction on the length of the object ID has been removed in this release.
reason
Indicates the event that caused this method to be invoked. The reason code can be one of the following:

DR_METHOD_END
Indicates that the object is being deactivated after the completion of a method. It is used if the object’s deactivation policy is:
- method
- transaction (only if there is no transaction in effect)
- process (if \texttt{TP::deactivateEnable()} called)

DR_SERVER_SHUTDOWN
Indicates that the object is being deactivated because the server is being shut down in an orderly fashion. It is used if the object’s deactivation policy is:
- transaction (only if transaction is active)
- process

Note that when a server is shut down in an orderly fashion, all transactions that the server is involved in are marked for rollback.

DR_TRANS_ABORTED
This reason code is used only for objects that have the transaction activation policy. It can occur when the transaction is started by either the client or automatically by the system. When the \texttt{deactivate_object()} method is invoked with this reason code, the transaction is marked for rollback only.

DR_TRANS_COMMITTING
This reason code is used only for objects that have the transaction activation policy. It can occur when the transaction is started by either the client or the TP Framework. It indicates that a \texttt{Current.commit()} operation was invoked for the transaction in which the object is involved. The \texttt{deactivate_object()} method is invoked just before the transaction manager’s two-phase commit algorithm begins; that is, before \texttt{prepare} is sent to the resource managers.

The CORBA object is allowed to vote on the outcome of the transaction when the \texttt{deactivate_object()} method is invoked with the DR_TRANS_COMMITTING reason code. By invoking \texttt{Current.rollback_only()}, the method can force the transaction to be rolled back; otherwise, the two-phase commit algorithm continues. The transaction is not necessarily committed just because the \texttt{Current.rollback_only()} is not invoked in this method. Any other CORBA object or resource manager involved in the transaction could also vote to roll back the transaction.
**DR_EXPLICIT_DEACTIVATE**

Indicates that the object is being deactivated because the application executed a TP::deactivateEnable(-,-,-) on this object. This can happen only for objects that have the process activation policy.

**Description**

Object deactivation is initiated either by the system or by the application, depending on the activation policy of the implementation for the CORBA object. The deactivate_object() method is invoked before the CORBA object is deactivated. For details of these policies and their use, see the section ICF Syntax.

Deactivation may occur after an execution of a method invoked by a client if the activation policy for the CORBA object implementation is method, or as a result of the end of transactional work if the activation policy is transaction. It may also occur as the result of server shutdown if the activation policy is transaction or process.

In addition, the BEA Tuxedo software supports the use of user-controlled deactivation of CORBA objects having an activation policy of process or method via the use of the TP::deactivateEnable() and TP::deactivateEnable(-,-,-) methods. TP::deactivateEnable can be called inside a method of an object to cause the object to be deactivated at the end of the method. If TP::deactivateEnable is called in an object with the transaction activation policy, an exception is raised (TobjS::IllegalOperation) and the TP Framework takes no action. TP::deactivateEnable(-,-,-) can be called to deactivate any object that has a process activation policy. For more information, see the section TP::deactivateEnable().

**Note:** The deactivate_object method will be called at server shutdown time for every object remaining in the Active Object Map, whether it was entered there implicitly by the TP Framework (the activation-on-demand technique: TP::create_servant and the servant’s activate_object method) or explicitly by the user with TP::create_active_object_reference.

The activate_object() and deactivate_object() methods and explicit methods invoked by the client can be used by the programmer to manage object state. The manner in which these methods are used to manage object state may vary according to the needs of the application. For a discussion of how these methods might be used, see Creating CORBA Server Applications.

The CORBA object with transaction activation policy gets to vote on the outcome of the transaction when the deactivate_object() method is invoked with the DR_TRANS_COMMITTING reason code. By calling Current.rollback_only() the method can force the transaction to be rolled back; otherwise, the two-phase commit algorithm continues.
transaction will not necessarily be committed just because `Current.rollback_only()` is not called in this method. Any other CORBA object or resource manager involved in the transaction could also vote to roll back the transaction.

**Restriction**

Note that if the object is involved in a transaction when this method is invoked, there are restrictions on what type of processing can be done based on the reason the object is invoked. If the object was involved in a transaction, the activation policy is `transaction` and the reason code for the call is:

- **DR_TRANS_ABORTED**
  
  No invocations on any CORBA objects are allowed in the method. No `tpcall()` is allowed. Transactions cannot be suspended or begun.

- **DR_TRANS_COMMITTING**
  
  No invocations on any CORBA objects are allowed in the method. No `tpcall()` is allowed. Transactions cannot be suspended or begun.

The reason for these restrictions is that the deactivation of objects with activation policy `transaction` is controlled by a call to the TP Framework from the transaction manager for the transaction. When the call with reason code `DR_TRANS_COMMITTING` is made, the transaction manager is executing phase 1 (prepare) of the two-phase commit. At this stage, it is not possible to issue a call to suspend a transaction nor to initiate a new transaction. Since a call to a CORBA object that was in another process would require that process to join the transaction, and the transaction manager is already executing the prepare phase, this would cause an error. Since a call to a CORBA object that had no transactional properties would require that the current transaction be suspended, this would also cause an error. The same is true of a `tpcall()`.

Similarly, when the invocation with reason code `DR_TRANS_ABORTED` is made, the transaction manager is already aborting. While the transaction manager is aborting, it is not possible to either suspend a transaction or initiate a new transaction. The same restrictions apply as for `DR_TRANS_COMMITTING`.

**Return Value**

None.

---

1. In theory, this would mean that an invocation on a transactional CORBA object in the same process would be valid since it would not require a new process to be registered with the transaction manager. However, it is not possible for the programmer to guarantee that an invocation on a CORBA object will occur in-proc, therefore, this practice is discouraged.
Exceptions

If the CORBA object method that is invoked by the client raises an exception, that exception is caught by the TP Framework and is eventually returned to the client. This is true even if deactivate_object() is invoked and raises an exception.

The client will never be notified about exceptions that are raised in deactivate_object(). It is the responsibility of the application code to check that the stored state of the CORBA object is consistent. For example, the application code could save a persistent flag that indicates whether or not deactivate_object() successfully saved the state. That flag can then be checked in activate_object().

If an error occurs while executing deactivate_object(), the application code should raise either a CORBA standard exception or a DeactivateObjectFailed exception. If deactivate_object() was invoked by the TP Framework, the TP Framework catches the exception and the following events occur:

- The object is deactivated.
- If the client initiated a transaction, the transaction is not rolled back.
- The client is not notified of the exception that is raised in deactivate_object().
- Based on which exception is raised, a message is logged to the user log (ULOG) file, as follows:

  TObject::DeactivateObjectFailed

  "TPFW_CAT:27: ERROR: De-activating object - application raised
  TObject::DeactivateObjectFailed. Reason = reason. Interface =
  interfaceName, OID = oid"

  Where reason is a user-supplied reason, and interfaceName and oid are the interface ID and object ID, respectively, of the invoked CORBA object.

  CORBA::Exception

  "TPFW_CAT:28: ERROR: De-activating object - CORBA Exception not
  handled by application. Exception ID = exceptionID. Interface =
  interfaceName, OID = oid"

  Where exceptionID is the interface ID of the exception, and interfaceName and oid are the interface ID and object ID, respectively, of the invoked CORBA object.

  Other exception

  "TPFW_CAT:29: ERROR: De-activating object - Unknown Exception not
  handled by application. Exception ID = exceptionID. Interface =
  interfaceName, OID = oid"
Where `exceptionID` is the interface ID of the exception, and `interfaceName` and `oid` are the interface ID and object ID, respectively, of the invoked CORBA object.

**Tobj_ServantBase::is_reentrant()**

**Synopsis**

Indicates that the object supports concurrent, reentrant invocations. This method supports the development of a multithreaded server application.

**C++ Binding**

```cpp
CORBA::Boolean _is_reentrant()
```

**Arguments**

None.

**Description**

The BEA Tuxedo server infrastructure calls this method to determine whether the servant implementation supports a reentrant invocation. To support reentrancy, a servant must include the necessary code to protect the integrity of its state while multiple threads interact with the object.

The `Tobj_ServantBase` class provides a default implementation of the `_is_reentrant` method that returns `FALSE`.

**Return Value**

CORBA::Boolean

Returns `TRUE` if the servant can support reentrancy.

**Example**

```cpp
CORBA::Boolean Simple_i::_is_reentrant() {
    TP::userlog("_is_reentrant called in thread %ld",
        (unsigned long)SIMPTHR_GETCURRENTTHREADID);
    return CORBA_TRUE;
}
```
**Tobj_ServantBase::_remove_ref()**

**Synopsis**

Releases a reference to a servant. This method supports the development of a multithreaded server application.

**Note:** In applications written using BEA Tuxedo release 8.0 or later, use this method instead of the C++ “delete” statement that you used previously with the TP::application_responsibility() method.

**C++ Binding**

```cpp
void _remove_ref()
```

**Parameters**

None.

**Description**

Invoke this method when a reference to a servant is no longer needed. Invoking this method causes the reference count for the servant to be decremented by one. If the _remove_ref() method brings the reference count to zero, it also calls the C++ “delete” statement on its own this pointer and deletes the servant.

**Return Value**

None.

**Example**

```cpp
if(servant != NULL)
    servant->_remove_ref();
```

**TP Interface**

The TP interface supplies a set of service methods that can be invoked by application code. This is the only interface in the TP Framework that can safely be invoked by application code. All other interfaces have callback methods that are intended to be invoked only by system code.

The purpose of this interface is to provide high-level calls that application code can call, instead of calls to underlying APIs provided by the Portable Object Adapter (POA), the CORBA Naming Service, and the BEA Tuxedo system. By using these calls, programmers can learn a simpler API
and are spared the complexity of the underlying APIs. The TP interface implicitly uses two features of the BEA Tuxedo software that extend the CORBA APIs:

- Factories and the FactoryFinder object
- Factory-based routing

For information about the FactoryFinder object, see the section FactoryFinder Interface. For more information about factory-based routing, see Setting Up a BEA Tuxedo Application.

**Usage Notes**

- During server application initialization, the application constructs the object reference for an application factory. It then invokes the register_factory() method, passing in the factory's object reference together with a factory id field. On server release (shutdown), the application uses the unregister_factory() method to unregister the factory.

- The TP class is a C++ native class.

- The TP.h file contains the declarations and definitions for the TP class.

**C++ Declarations (in TP.h)**

The C++ mapping is as follows:

```cpp
class TP {
public:
    static CORBA::Object_ptr create_object_reference(
        const char* interfaceName,
        const char* stroid,
        CORBA::NVList_ptr criteria);

    static CORBA::Object_ptr create_active_object_reference(
        const char* interfaceName,
        const char* stroid,
        Tobj_Servant servant);

    static CORBA::Object_ptr get_object_reference();

    static void register_factory(
        CORBA::Object_ptr factory_or,
        const char* factory_id);

    static void unregister_factory(
        CORBA::Object_ptr factory_or,
        const char* factory_id);

    static void deactivateEnable();
};
```
static void deactivateEnable(
    const char* interfaceName,
    const char* stroid,
    Tobj_Servant servant);

static CORBA::ORB_ptr orb();
static Tobj_Bootstrap* bootstrap();
static void open_xa_rm();
static void close_xa_rm();
static int userlog(char*, ...);
static char* get_object_id(CORBA::Object_ptr obj);
static void application_responsibility(
    Tobj_Servant servant);
};

TP::application_responsibility()

Synopsis
Tells the TP Framework that the application is taking responsibility for the servant’s lifetime.

Note: Do not use this method in applications written using BEA Tuxedo release 8.0 or later; instead, use the Tobj_ServantBase::_add_ref() method.

C++ Binding
static void application_responsibility(Tobj_Servant servant);

Arguments
    servant
A pointer to a servant that is already known to the TP Framework.

Exceptions
    TobjS::InvalidServant
    Indicates that the specified servant is NULL.

Description
This method tells the TP Framework that the application is taking responsibility for the servant’s lifetime. As a result of this call, when the TP Framework has completed deactivating the object (that is, after invoking the servant’s deactivate_object method), the TP Framework does nothing more with the object.
Once an application has taken responsibility for a servant, the application must take care to delete servant when it is no longer needed, the same as for any other C++ instance.

If the servant is not known to the TP Framework (that is, it is not active), this call has no effect.

Return Values
None.

TP::bootstrap()

Synopsis
Returns a pointer to a Tobj::Tobj_Bootstrap object. The Bootstrap object is used to access initial object references for the FactoryFinder object, the Interface Repository, the TransactionCurrent, and the SecurityCurrent objects.

C++ Binding

static Tobj_Bootstrap* TP::bootstrap();

Arguments
None.

Return Value
Upon successful completion, bootstrap() returns a pointer to the Tobj::Tobj_Bootstrap object that is created by the TP Framework when the server application is started.

Exceptions
None.

Description
The TP Framework creates a Tobj::Tobj_Bootstrap object as part of initialization; it is not necessary for the application code to create any other Tobj::Tobj_Bootstrap objects in the server.

Caution: Because the TP Framework owns the Tobj::Tobj_Bootstrap object, server application code must not dispose of the Bootstrap object.

Note: If you are using the CORBA INS bootstrap mechanism and you are not using the SecurityCurrent for security or TransactionCurrent for transactions, you do not need to use the Bootstrap object.
TP::close_xa_rm()

Synopsis
Closes the XA resource manager to which the invoking process is linked.

C++ Binding
static void TP::close_xa_rm();

Arguments
None.

Description
The close_xa_rm() method closes the XA resource manager to which the invoking process is linked. XA resource managers are provided by database vendors, such as Oracle and Informix.

Note: The functionality of this call is also provided by Tobj::TransactionCurrent::close_xa_rm(). The TP::close_xa_rm() method provides a more convenient way for a server application to close a resource manager because there is no need to obtain an object reference to the TransactionCurrent object. A reference to the TransactionCurrent object can be obtained from the Bootstrap object. See TP::bootstrap() for an explanation of how to obtain a reference to the Bootstrap object. For more information about the TransactionCurrent object, see the CORBA Bootstrapping Programming Reference section and Using CORBA Transactions.

This method should be invoked once from the Server::release() method for each server that is involved in global transactions. This includes servers that are linked with an XA resource manager, as well as servers that are involved in global transactions, but are not actually linked with an XA-compliant resource manager.

The close_xa_rm() method should be invoked in place of a close invocation that is specific to the resource manager. Because resource managers differ in their initialization semantics, the specific information needed to close a particular resource manager is placed in the CLOSEINFO parameter in the GROUPS section of the BEA Tuxedo system UBBCONFIG file.

The format of the CLOSEINFO string is dependent on the requirements of the database vendor providing the underlying resource manager. For more information about the CLOSEINFO parameter, see Setting Up a BEA Tuxedo Application and the ubbconfig(5) reference page in the File Formats, Data Descriptions, MIBs, and System Processes Reference. Also, refer to database vendor documentation for information about how to develop and install applications that use the XA libraries.
Return Values

None.

Exceptions

CORBA::BAD_INV_ORDER
There is an active transaction. The resource manager cannot be closed while a transaction is active.

Tobj::RMFailed
The tx_close() call returned an error return code.

Note: Unlike other exceptions returned by the TP Framework, the Tobj::RMFailed exception is defined in tobj_c.h (which is derived from tobj.idl), not TobjS_c.h (which is derived from TobjS.idl). This is because native clients can also open XA resource managers. Therefore, the exception returned is consistent with the exception expected by native client code and by Server::release() if it uses the alternate mechanism, TransactionCurrent::close_xa_rm, which is shared with native clients.

TP::create_active_object_reference()

Synopsis

Creates an object reference and preactivates an object.

C++ Binding

static CORBA::Object_ptr
create_active_object_reference(
    const char*       interfaceName,
    const char*      stroid,
    Tobj_Servant     servant);

Arguments

interfaceName
Specifies a character string that contains the fully qualified interface name for the object.

stroid
Specifies the ObjectId in string format. The ObjectId uniquely identifies this instance of the class. The programmer decides what information to place in the ObjectId. One possibility would be to use it to hold a database key. Choosing the value of an object identifier, and the degree of uniqueness, is part of the application design. The BEA Tuxedo software cannot guarantee any uniqueness in object references, since these may
be legitimately copied and shared outside the BEA Tuxedo environment, for example by stringifying the object reference.

servant
A pointer to a servant that the application has already created and initialized.

Exceptions:

TobjS::InvalidInterface
Indicates that the specified interfaceName is NULL.

TobjS::InvalidObjectId
Indicates the specified stroid is NULL.

TobjS::ServantAlreadyActive
The object could not be activated explicitly because the servant is already being used with another ObjectId. A servant can be used only with a single ObjectId. To preactivate objects containing different ObjectIds, the application must create multiple servants and preactivate them separately, one per ObjectId.

TobjS::ObjectAlreadyActive
The object could not be activated explicitly because the ObjectId is already being used in the Active Object Map. A given ObjectId can have only one servant associated with it. To change to a different servant, the application must first deactivate the object and activate it again.

TobjS::IllegalOperation
The object could not be activated explicitly because it does not have the process activation policy.

Description
This method creates an object reference and preactivates an object. The resulting object reference may be passed to clients who will use it to access the object.

Ordinarily, the application will call this method in two places:

- In Server::initialize() to preactivate process objects so that they do not need activation on the first invocation.

- In any method that creates object references to be returned to clients.

This method allows an application to activate an object explicitly before its first invocation. (For reasons you might want to do this, refer to the section Explicit Activation.) The user first creates a servant and sets its state before calling create_active_object_reference. The TP Framework then enters the servant and strings the ObjectId in the Active Object Map. The result is
exactly the same as if the TP Framework had previously invoked `Server::create_servant`, received back the servant pointer, and then had invoked `servant::activate_object`.

The object so activated must be for an interface that was declared with the process activation policy; otherwise, an exception is raised.

If the object is deactivated, an object reference held by a client might cause the object to be activated again in some other process. For a discussion about situations in which this might be a problem, refer to the section Explicit Activation.

**Note:** There is one restriction on this method when the user-controlled concurrency policy option is set in the ICF file (See Parallel Objects). The `TP::create_active_object_reference` method throws a `TobjS::IllegalOperation` exception if it is passed an interface with user-controlled concurrency set. Since the AOM is not used when user-controlled concurrency is set, there is no way for the TP Framework to connect an active object to this server.

**Caution**

When you preactivate objects in an interface, you must specify an activation policy of process in the ICF file for that interface. However, when you specify the process activation policy for an interface in the ICF file, this can lead to the following problem.

**Problem Statement**

1. You write SERVER1 such that all objects on interface A are preactivated. To prevent the object from being activated on demand by the TP Framework, you write the interface's `activate_object` method to always throw the `ActivateObjectFailed` exception.

2. SERVER2 also implements objects of interface A. However, instead of preactivating the objects, SERVER2 lets the TP Framework activate them on demand.

3. If the administrator configures SERVER1 and SERVER2 in the same group, then a client can get an interface A object reference from SERVER2 and invoke on it. Then, due to load balancing, SERVER1 could be asked to activate an object on interface A. However, SERVER1 is not able to activate an object on interface A on demand because its `activate_object` method throws the `ActivateObjectFailed` exception.

**Workaround**

You can avoid this problem by having the administrator configure SERVER1 and SERVER2 in different groups. The administrator uses the `SERVERS` section of the `UBBCONFIG` file to define groups.
Return Value
The newly created object reference.

**TP::create_object_reference()**

Synopsis
Creates an object reference. The resulting object reference may be passed to clients who use it to access the object.

C++ Binding
```
static CORBA::Object_ptr TP::create_object_reference (
    const char* interfaceName,
    const char* stroid,
    CORBA::NVList_ptr criteria);
```

Arguments

- **interfaceName**
  Specifies a character string that contains the fully qualified interface name for the object. The interface name can be retrieved by making a call on the following interface typecode ID function:
  ```
  const char* _tc_<CORBA interface name>::id();
  ```
  where `<CORBA interface name>` is any object class name. For example:
  ```
  char* idiname = _tc_Simple->id();
  ```

- **stroid**
  Specifies the ObjectId in string format. The ObjectId uniquely identifies this instance of the class. It is up to the programmer to decide what information to place in the ObjectId. One possibility would be to use the ObjectId to hold a database key. Choosing the value of an object identifier, and the degree of uniqueness, is part of the application design. The BEA Tuxedo software cannot guarantee any uniqueness in object references, since object references may be legitimately copied and shared outside the BEA Tuxedo domain (for example, by passing the object reference as a string). It is strongly recommended the you choose a unique ObjectId in order to allow parallel execution of invokes on object references.

Note: The restriction on the length of the ObjectId has been removed in this release.
criteria
Specifies a list of named values that can be used to provide factory-based routing for the object reference. The list is optional and is of type CORBA::NVList. The use of factory-based routing is optional and is dependent on the use of this argument. If you do not want to use factory-based routing, you can pass a value of 0 (zero) for this argument. The BEA Tuxedo system administrator configures factory-based routing by specifying routing rules in the UBBCONFIG file. See Setting Up a BEA Tuxedo Application online document for details on this facility.

Exceptions
The following exceptions can be raised by the create_object_reference() method:

InvalidInterface
Indicates that the specified interfaceName is NULL.

InvalidObjectId
Indicates that the specified stroid is NULL.

Description
The server application is responsible for invoking the create_object_reference() method. This method creates an object reference. The resulting object reference may be passed to clients who will use it to access the object.

Ordinarily, the server application calls this method in two places:

- In Server::initialize() to create factories for the server.
- In factory methods to create object references to be returned to clients.

For examples of how and when to call the create_object_reference() method, see Creating CORBA Server Applications.

Return Value
Object
The newly created object reference.

Example
The following example shows how to use the criteria argument:

CORBA::NVList_ptr criteria;
CORBA::Long branch_id = 7;
CORBA::Long account_id = 10001;
CORBA::Any any_val;
// Create the list and assign to _var to cleanup on exit
CORBA::ORB::create_list (2, criteria);
CORBA::NVList_var criteria_var(criteria);

// Add the BRANCH_ID
any_val <<= branch_id;
criteria->add_value("BRANCH_ID", any_val, 0);

// Add the ACCOUNT_ID
any_val <<= account_id;
criteria->add_value("ACCOUNT_ID", any_val, 0);

// Create the object reference.
TP::create_object_reference ("IDL:BankApp/Teller:1.0",
"Teller_01", criteria);

TP::deactivateEnable()

Synopsis
Enables application-controlled deactivation of CORBA objects.

C++ Binding
Current-object format:
static void           TP::deactivateEnable();

Any-object format:
static void        TP::deactivateEnable(
    const char*  interfaceName,
    const char* stroid,
    Tobj_Servant servant);

Arguments

interfaceName
   Specifies a character string that contains the fully qualified interface name for the object.

stroid
   Specifies the ObjectId in string format for the object to be deactivated.

servant
   A pointer to the servant associated with the stroid.
Exceptions

The following exceptions can be raised by the `deactivateEnable()` method:

**IllegalOperation**
Indicates that the `TP::deactivateEnable` method was invoked by an object with the activation policy set to `transaction`.

**TobjS::ObjectNotActive**
In the Any-object format, the object specified could not be deactivated because it was not active (the `stroid` and `servant` parameters did not identify an object that was in the Active Object Map).

Description

This method can be used to cause deactivation of an object, either the object currently executing (upon completion of the method in which it is called) or another object. It can only be used for objects with the process activation policy. It provides additional flexibility for objects with the process activation policy.

**Note:** For single-threaded servers, the `TP::deactivateEnable(interface, object id, servant)` method can be used to deactivate an object. However, if that object is currently in a transaction, the object will be deactivated when the transaction commits or rolls back. If an invoke occurs on the object before the transaction is committed or rolled back, the object will not be deactivated.

To ensure the desired behavior, make sure that the object is not in a transaction or ensure that no invokes occur on the object after the `TP::deactivateEnable()` call until the transaction is complete.

**Note:** For multithreaded servers, use of the `TP::deactivateEnable(interface, object id, servant)` method is not supported for deactivation of objects in per-object servers. This method is supported for deactivation objects in per-request servers, however, the deactivation may be delayed because others threads are acting on the object.

Depending on which of the overloaded functions are called, the actions are as follows.

**Current-object format**
When called from within a method of an object with process activation policy, the object currently executing will be deactivated after completing the method being executed.

When called from within a method of an object with method activation, the effect is the same as the normal behavior of such objects (effectively, a NOOP).

When the object is deactivated, the TP Framework first removes the object from the Active Object Map. It then calls the associated servant’s `deactivate_object` method with a reason of `DR_METHOD_END`. 
**Any-object format**

The application can request deactivation of an object by specifying its ObjectId and the associated servant.

If the object is currently executing, the TP Framework marks it for deactivation and waits until the object’s method completes before deactivating the object (as with the “current-object format”). If the object is not currently executing, the TP Framework may deactivate it immediately. No indication is given to the caller as to the status of the deactivation. When the object is deactivated, the TP Framework first removes the object from the Active Object Map. It then calls the associated servant’s deactivate_object method with a reason of DR_EXPLICIT_DEACTIVATE.

If the object for which the deactivate is requested has a transaction activation policy, an IllegalOperation exception is raised. This is because deactivation of such objects may interfere with their correct notification of transaction completion by the BEA Tuxedo transaction manager.

**Return Value**

None.

**TP::get_object_id()**

**Synopsis**

Allows a server to retrieve the string ObjectId contained in an object reference that was created in the TP Framework.

**C++ Binding**

```cpp
char* TP::get_object_id(Corba::Object_ptr obj);
```

**Arguments**

- **obj**

  The object reference from which to get the ObjectId.

**Exception**

- TobjS::InvalidObject

  The object is nil or was not created by the TP Framework.
Description

This method allows a server to retrieve the string ObjectId contained in an object reference that was created in the TP Framework. If the object reference was not created in the TP Framework (for example, it was created by a client ORB), an exception is raised.

The caller must call CORBA::string_free on the returned value when the object reference is no longer needed.

Return Value

The string ObjectId passed to TP::create_object_reference or TP::create_active_object_reference when the object reference was created.

TP::get_object_reference()

Synopsis

Returns a pointer to the current object.

C++ Binding

static CORBA::Object_ptr TP::get_object_reference();

Arguments

None.

Note that if get_object_reference() is invoked from within either Server::initialize() or Server::release(), it is considered to be invoked outside the scope of an application’s TP object execution; therefore, the TobjS::NilObject exception is raised.

Exceptions

The following exception can be raised by the get_object_reference() method:

NilObject

Indicates that the method was invoked outside the scope of an application’s CORBA object execution. The reason string contains OutOfScope.

Description

This method returns a pointer to the current object. The CORBA::Object_ptr pointer that is returned can be passed to a client.
Return Value

The `get_object_reference()` method returns a `CORBA::Object_ptr` for the current object when invoked within the scope of a CORBA object execution. Otherwise, the `TobjS::NilObject` exception is raised.

**TP::open_xa_rm()**

Synopsis

Opens the XA resource manager to which the invoking process is linked.

C++ Binding

```cpp
static void TP::open_xa_rm();
```

Arguments

None.

Exceptions

`Tobj::RMFailed`

The `tx_open()` call returned an error return code.

**Note:** Unlike other exceptions returned by the TP Framework, this exception is defined in `tobj_c.h` (which is derived from `tobj.idl`), not in `TobjS_c.h` (which is derived from `TobjS.idl`). This is because native clients can also open XA resource managers. Therefore, the exception returned is consistent with the exception expected by native client code and by `Server::release()` if it uses the alternate mechanism, `TransactionCurrent::close_xa_rm`, which is shared with native clients.

Description

The `open_xa_rm()` method opens the XA resource manager to which the invoking process is linked. XA resource managers are provided by database vendors, such as Oracle and Informix.

**Note:** The functionality of this method is also provided by

`Tobj::TransactionCurrent::close_xa_rm()`. However, `TP::open_xa_rm()` provides a more convenient way for a server application to close a resource manager because there is no need to obtain an object reference to the TransactionCurrent object. A reference to the TransactionCurrent object can be obtained from the Bootstrap object. See `TP::bootstrap()` for an explanation of how to obtain a reference to the Bootstrap object. For more information about the TransactionCurrent object, see the CORBA Bootstrapping Programming Reference section and *Using CORBA Transactions*.
This method should be invoked once from the Server::initialize() method for each server that participates in a global transaction. This includes servers that are linked with an XA resource manager, as well as servers that participate in a global transaction, but are not actually linked with an XA-compliant resource manager.

The open_xa_rm() method should be invoked in place of an open invocation that is specific to a resource manager. Because resource managers differ in their initialization semantics, the specific information needed to open a particular resource manager is placed in the OPENINFO parameter in the GROUPS section of the UBB_CONFIG file.

The format of the OPENINFO string is dependent on the requirements of the database vendor providing the underlying resource manager. For more information about the CLOSEINFO parameter, see Setting Up a BEA Tuxedo Application and the ubbconfig(5) reference page in the File Formats, Data Descriptions, MIBs, and System Processes Reference. Also, refer to database vendor documentation for information about how to develop and install applications that use the XA libraries.

Note: Only one resource manager can be linked to the invoking process.

Return Values
None.

TP::orb()

Synopsis
Returns a pointer to an ORB object.

C++ Binding
static CORBA::ORB_ptr TP::orb();

Arguments
None.

Exceptions
None.

Description
Access to the ORB object allows the application to invoke ORB operations, such as string_to_object() and object_to_string().

Note: Because the TP Framework owns the ORB object, the application must not delete it.

Return Value

Upon successful completion, orb() returns a pointer to the ORB object that is created by the TP Framework when the server program is started.

TP::register_factory()

Synopsis

Locates the BEA Tuxedo FactoryFinder object and registers a BEA Tuxedo factory.

C++ Binding

static void TP::register_factory(
    CORBA::Object_ptr factory_or, const char* factory_id);

Arguments

factory_or

Specifies the object reference that was created for an application factory using the TP::create_object_reference() method.

factory_id

Specifies a string identifier that is used to identify the application factory. For some suggestions as to the composition of this string, see Creating CORBA Server Applications.

Exceptions

The following exceptions can be raised by the register_factory() method:

TobjS::CannotProceed

Indicates that the FactoryFinder encountered an internal error during the search, with the error being written to the user log (ULOG). Notify the operations staff immediately if this exception is raised. Depending on the severity of the internal error, the server running the FactoryFinder or the NameManager may have terminated. If a FactoryFinder service has terminated, start a new FactoryFinder service. If the NameManager has terminated, and there is another NameManager running, start a new one. If no NameManagers are running, restart the application.

TobjS::InvalidName

Indicates that the id string is empty. It is also raised if the field contains blank spaces or control characters.
TobjS::InvalidObject
   Indicates that the factory value is nil.

TobjS::RegistrarNotAvailable
   Indicates that the FactoryFinder object cannot locate the NameManager. Notify the
   operations staff immediately if this exception is raised. If no naming services servers are
   running, restart the application.

   Note: Another possible reason that this exception might occur is that the FactoryFinder
   cannot participate in a transaction. Therefore, you may need to suspend the current
   transaction before issuing the TP::register_factory() and
   TP::unregister_factory() calls. For information on suspending and resuming
   transactions, see Using CORBA Transactions in the online documentation.

TobjS::OverFlow
   Indicates that the id string is longer than 128 bytes (currently the maximum allowable
   length).

Description
   This method locates the BEA Tuxedo FactoryFinder object and registers a BEA Tuxedo factory.
   Typically, TP::register_factory() is invoked from Server::initialize() when the
   server creates its factories. The register_factory() method locates the BEA Tuxedo
   FactoryFinder object and registers the BEA Tuxedo factory.

   Caution: Callback objects (that is, those created by a joint client/server directly through the
   POA) should not be registered with a FactoryFinder.

Return Value
   None.

TP::unregister_factory()

Synopsis
   Locates the BEA Tuxedo FactoryFinder object and removes a factory.

C++ Binding
   static void TP::unregister_factory ( 
       CORBA::Object_ptr factory_or, const char* factory_id);
Arguments

factory_or
Specifies the object reference that was created for an application factory using the
TP::create_object_reference() method.

factory_id
Specifies a string identifier that is used to identify the application factory. For some
suggestions as to the composition of this string, see Creating CORBA Server Applications.

Exceptions

The following exceptions can be raised by the unregister_factory() method:

CannotProceed
Indicates that the FactoryFinder encountered an internal error during the search, with the
error being written to the user log (ULOG). Notify the operations staff immediately if this
exception is raised. Depending on the severity of the internal error, the server running the
FactoryFinder or the NameManager may have terminated. If a FactoryFinder service has
terminated, start a new FactoryFinder service. If the NameManager has terminated, and
there is another NameManager running, start a new one. If no NameManagers are running,
restart the application.

InvalidName
Indicates that the id string is empty. It is also raised if the field contains blank spaces or
control characters.

RegistrarNotAvailable
Indicates that the FactoryFinder object cannot locate the NameManager. Notify the
operations staff immediately if this exception is raised. If no naming services servers are
running, restart the application.

Note: Another possible reason that this exception might occur is that the FactoryFinder
cannot participate in a transaction. Therefore, you may need to suspend the current
transaction before issuing the TP::register_factory() and
TP::unregister_factory() calls. For information on suspending and resuming
transactions, see Using CORBA Transactions in the online documentation.

TobjS::OverFlow
Indicates that the id string is longer than 128 bytes (currently the maximum allowable
length).
Description

This method locates the BEA Tuxedo FactoryFinder object and removes a factory. Typically
TP::unregister_factory() is invoked from Server::release() to unregister server
factories.

Return Value

None.

TP::userlog()

Synopsis

Writes a message to the user log (ULOG) file.

C++ Binding

static int TP::userlog(char*, ...);

Arguments

The first argument is a printf(3S) style format specification. The printf(3S) argument is
described in a C or C++ reference manual.

Exceptions

None.

Description

The userlog() method writes a message to the user log (ULOG) file. Messages are appended to
the ULOG file with a tag made up of the time (hhmms), system name, process name, and
process-id of the invoking process. The tag is terminated with a colon.

We recommend that server applications limit their use of userlog() messages to messages that
can be used to help debug application errors; flooding the ULOG file with incidental information
make it difficult to spot actual errors.

Return Value

The userlog() method returns the number of characters that were output, or a negative value if
an output error was encountered. Output errors include the inability to open or write to the current
log file.
Example

The following example shows how to use the `TP::userlog()` method:

```c
userlog ("System exception caught: %s", e.get_id());
```

**CosTransactions::TransactionalObject Interface Not Enforced**

Use of this interface is now deprecated. Therefore, the use of this interface is now optional and no enforcement of descent from this interface is done for objects infected with transactions. The programmer can specify that an object is not to be infected by transactions by specifying the `never` or `ignore` transaction policies. There is no interface enforcement for eligibility for transactions. The only indicator is the transaction policy.

**Note:** The CORBA services Object Transaction Service does not require that all requests be performed within the scope of a transaction. It is up to each object to determine its behavior when invoked outside the scope of a transaction; an object that requires a transaction context can raise a standard exception.

**Error Conditions, Exceptions, and Error Messages**

**Exceptions Raised by the TP Framework**

The following exceptions are raised by the TP Framework and are returned to clients when error conditions occur in, or are detected by, the TP Framework:

- `CORBA::INTERNAL`
- `CORBA::OBJECT_NOT_EXIST`
- `CORBA::OBJ_ADAPTER`
- `CORBA::INVALID_TRANSACTION`
- `CORBA::TRANSACTION_ROLLED_BACK`

Since the reason for these exceptions may be ambiguous, each time one of these exceptions is raised, the TP Framework also writes a descriptive error message that explains the reason to the user log file.

**Exceptions in the Server Application Code**

Exceptions raised within a method invoked by a client are always raised back to the client exactly as they were raised in the method invoked by the client.
The following TP Framework callback methods are initiated by events other than client requests on the object:

Tobj_ServantBase::activate_object()
Tobj_ServantBase::deactivate_object()
Server::create_servant()

If exception conditions are raised in these methods, those exact exceptions are not reported back to the client. However, each of these methods is defined to raise an exception that includes a reason string. The TP Framework will catch the exception raised by the callback and log the reason string to the user log file. The TP Framework may raise an exception back to the client. Refer to the descriptions of the individual TP Framework callback methods for more information about these exceptions.

Example

For Tobj_ServantBase::deactivate_object(), the following line of code throws a DeactivateObjectFailed exception:

```
throw TobjS::DeactivateObjectFailed( "deactivate failed to save state!");
```

This message is appended to the user log file with a tag made up of the time (hhmms), system name, process name, and process-id of the calling process. The tag is terminated with a colon. The preceding throw statement causes the following line to appear in the user log file:

```
151104.T1!simpapps.247: APPEXC: deactivate failed to save state!
```

Where 151104 is the time (3:11:04pm), T1 is the system name, simpapps is the process name, 247 is the process-id, and APPEXC identifies the message as an application exception message.

Exceptions and Transactions

Exceptions that are raised in either CORBA object methods or in TP Framework callback methods will not automatically cause a transaction to be rolled back unless the TP Framework started the transaction. It is up to the application code to call Current.rollback_only() if the condition that caused the exception to be raised should also cause the transaction to be rolled back.

Restriction of Nested Calls on CORBA Objects

The TP Framework restricts nested calls on CORBA objects. The restriction is as follows:
During a client invocation of a method of CORBA object A, CORBA object A cannot be invoked by another CORBA object B that is acting as a client of CORBA object A.

The TP Framework will detect the fact that a second CORBA object is acting as a client to an object that is already processing a method invocation, and will return a CORBA::OBJ_ADAPTER exception to the caller.

**Note:** Application code should not depend on this behavior; that is, users should not make any processing dependent on this behavior. This restriction may be lifted in a future release.
CHAPTER 4

CORBA Bootstrapping Programming Reference

This topic includes the following sections:

- Why Bootstrapping Is Needed
- Supported Bootstrapping Mechanisms
- BEA Bootstrapping Mechanism
- Bootstrap Object API
- Bootstrap Object Programming Examples
- Interoperable Naming Service Bootstrapping Mechanism

Note: The BEA Tuxedo CORBA Java client and BEA Tuxedo CORBA Java client ORB were deprecated in Tuxedo 8.1 and are no longer supported in Tuxedo 9.x. All BEA Tuxedo CORBA Java client and BEA Tuxedo CORBA Java client ORB text references, associated code samples, etc. should only be used:

- to help implement/run third party Java ORB libraries, and
- for programmer reference only.

Technical support for third party CORBA Java ORBs should be provided by their respective vendors. BEA Tuxedo does not provide any technical support or documentation for third party CORBA Java ORBs.
Why Bootstrapping Is Needed

To communicate with BEA Tuxedo objects, a client application must obtain object references. Without an object reference, there can be no communication. To solve this problem, client applications use a bootstrapping mechanism to obtain object references to objects in a BEA Tuxedo domain.

Supported Bootstrapping Mechanisms

In the Tuxedo 8.0 release and later, two bootstrapping mechanisms are supported:

- **BEA Bootstrapping Mechanism**
  Use this mechanism if you using the BEA client ORB.

- **Interoperable Naming Service Bootstrapping Mechanism**
  Use this mechanism if you using a client ORB from another vendor.

**Note:** The CORBA C++ client provided with BEA Tuxedo software may use the Interoperable Naming Service bootstrapping mechanism, however, for performance reasons, this is not recommended.

BEA Bootstrapping Mechanism

The BEA bootstrapping mechanism uses the Bootstrap object. Bootstrap objects are local programming objects, not remote CORBA objects, in both the client and the server. When Bootstrap objects are created, their constructor requires the network address of a BEA Tuxedo IIOP Listener/Handler. Given this information, the bootstrapping object can generate object references for the key remote objects in the BEA Tuxedo domain. These object references can then be used to access services available in the BEA Tuxedo domain.

How Bootstrap Objects Work

Bootstrap objects are created by a client or a server application that must access object references to the following BEA Tuxedo CORBA interfaces:

- FactoryFinder
- Security
- Interface Repository
Bootstrap objects may represent the first connection to a specific BEA Tuxedo domain depending on the format of the IIOP Listener/Handler address. If the NULL scheme Universal Resource Locator (URL) format is used (the only address format supported in releases of BEA WebLogic Enterprise prior to version 5.1 and BEA Tuxedo release 8.0), the Bootstrap objects represent the first connection. However, if the URL format is used, the connection will not occur until after creation of the Bootstrap object. For more information on address formats and connection times, refer to Tobj_Bootstrap.

For a BEA Tuxedo CORBA remote client, Bootstrap objects are created with the host and the port for the BEA Tuxedo IIOP Listener/Handler. However, for BEA Tuxedo native client and server applications, there is no need to specify a host and port because they execute in a specific BEA Tuxedo domain. The IIOP Listener/Handler host and the port ID are included in the BEA Tuxedo domain configuration information.

After they are created, Bootstrap objects satisfy requests for object references for objects in a particular BEA Tuxedo domain. Different Bootstrap objects allow the application to use multiple domains.

Using the Bootstrap object, you can obtain references to the following objects:

- **SecurityCurrent**
  
  The SecurityCurrent object is used to establish a security context within a BEA Tuxedo domain. The client can then obtain the PrincipalAuthenticator from the principal_authenticator attribute of the SecurityCurrent object.

- **TransactionCurrent**
  
  The TransactionCurrent object is used to participate in a BEA Tuxedo transaction. The basic operations are as follows:
  
  - **Begin**
    
    Begin a transaction. Future operations take place within the scope of this transaction.
  
  - **Commit**
End the transaction. All operations on this client application have completed successfully.

- Roll back
  Abort the transaction. Tell all other participants to roll back.
- Suspend
  Suspend participation in the current transaction. This operation returns an object that identifies the transaction and allows the client application to resume the transaction later.
- Resume
  Resume participation in the specified transaction.

**FactoryFinder**

The FactoryFinder object is used to obtain a factory. In the BEA Tuxedo system, factories are used to create application objects. The FactoryFinder provides the following different methods to find factories:

- Get a list of all available factories that match a factory object reference (find_factories).
- Get the factory that matches a name component consisting of id and kind (find_one_factory).
- Get the first available factory of a specific kind (find_one_factory_by_id).
- Get a list of all available factories of a specific kind (find_factories_by_id).
- Get a list of all registered factories (list_factories).

**InterfaceRepository**

The Interface Repository contains the interface descriptions of the CORBA objects that are implemented within the BEA Tuxedo domain. Clients using the Dynamic Invocation Interface (DII) need a reference to the Interface Repository to be able to build CORBA request structures.

**NamingService**

A NamingService object is used to obtain a reference to the root namespace. When you use this object, the ORB locates the root of the namespace.

**NotificationService**
The NotificationService object is used to obtain a reference to the event channel factory (CosNotifyChannelAdmin::EventChannelFactory) in the CosNotification Service. In the BEA Tuxedo system, the EventChannelFactory is used to locate the Notification Service channel.

- **Tobj_SimpleEventsService**
  
  The Tobj_SimpleEventsService object is used to obtain a reference to the event channel factory (Tobj_SimpleEvents::ChannelFactory) in the BEA Simple Events Service. In the BEA Tuxedo system, the ChannelFactory is used to locate the BEA Simple Events Service channel.

Using the bootstrapping mechanism, you can obtain six different references, as follows:

- **SecurityCurrent**

  The SecurityCurrent object is used to establish a security context within a BEA Tuxedo domain. The client can then obtain the PrincipalAuthenticator from the `principal_authenticator` attribute of the SecurityCurrent object.

- **TransactionCurrent**

  The TransactionCurrent object is used to participate in a BEA Tuxedo transaction. The basic operations are as follows:

  - **Begin**
    
    Begin a transaction. Future operations take place within the scope of this transaction.
  
  - **Commit**
    
    End the transaction. All operations on this client application have completed successfully.
  
  - **Roll back**
    
    Abort the transaction. Tell all other participants to roll back.
  
  - **Suspend**
    
    Suspend participation in the current transaction. This operation returns an object that identifies the transaction and allows the client application to resume the transaction later.
  
  - **Resume**
    
    Resume participation in the specified transaction.

- **FactoryFinder**

---

**CORBA Programming Reference**
The FactoryFinder object is used to obtain a factory. In BEA Tuxedo CORBA, factories are used to create application objects. The FactoryFinder provides the following different methods to find factories:

- Get a list of all available factories that match a factory object reference (find_factories).
- Get the factory that matches a name component consisting of id and kind (find_one_factory).
- Get the first available factory of a specific kind (find_one_factory_by_id).
- Get a list of all available factories of a specific kind (find_factories_by_id).
- Get a list of all registered factories (list_factories).

**InterfaceRepository**

The Interface Repository contains the interface descriptions of the CORBA objects that are implemented within the BEA Tuxedo domain. Clients using the Dynamic Invocation Interface (DII) need a reference to the Interface Repository to be able to build CORBA request structures. The ActiveX Client is a special case of this. Internally, the implementation of the COM/IIOP Bridge uses DII, so it must get the reference to the Interface Repository, although this is transparent to the desktop client.

**NotificationService**

The NotificationService object is used to obtain a reference to the event channel factory (CosNotifyChannelAdmin::EventChannelFactory) in the CosNotification Service. In BEA Tuxedo CORBA, the EventChannelFactory is used to locate the Notification Service channel.

**Tobj_SimpleEventsService**

The Tobj_SimpleEventsService object is used to obtain a reference to the event channel factory (Tobj_SimpleEvents::ChannelFactory) in the BEA Simple Events Service. In BEA Tuxedo CORBA, the ChannelFactory is used to locate the BEA Simple Events Service channel.

The FactoryFinder and Interface Repository objects are not implemented in the environmental objects library. However, they are specific to a BEA Tuxedo domain and are thus conceptually similar to the SecurityCurrent and TransactionCurrent objects in use.

The Bootstrap object implies an association or “session” between the client application and the BEA Tuxedo domain. Within the context of this association, the Bootstrap object imposes a containment relationship with the other Current objects (or contained objects); that is, the
SecurityCurrent and TransactionCurrent. Current objects are valid only for this domain and only while the Bootstrap object exists.

**Note:** Resolving the SecurityCurrent when using the new URL address format (corbaloc://hostname:port_number) is a local operation; that is, no connection is made by the client to the IIOP Listener/Handler.

In addition, a client can have only one instance of each of the Current objects at any time. If a Current object already exists, an attempt to create another Current object does not fail. Instead, another reference to the already existing object is handed out; that is, a client application may have more than one reference to the single instance of the Current object.

To create a new instance of a Current object, the application must first invoke the destroy_current() method on the Bootstrap object. This invalidates all of the Current objects, but does not destroy the session with the BEA Tuxedo domain. After invoking destroy_current(), new instances of the Current objects can be created within the BEA Tuxedo domain using the existing Bootstrap object.

To obtain Current objects for another domain, a different Bootstrap object must be constructed. Although it is possible to have multiple Bootstrap objects at one time, only one Bootstrap object may be “active;” that is, have Current objects associated with it. Thus, an application must first invoke destroy_current() on the “active” Bootstrap object before obtaining new Current objects on another Bootstrap object, which then becomes the active Bootstrap object.

**Note:** If you want to access objects in multiple domains, either import the object to the local domain or administratively configure your application access multiple domains. For more information on multi-domain configurations, see “Configuring Multiple CORBA Domains” in Using the BEA Tuxedo Domains Component.

Servers and native clients are inside of the BEA Tuxedo domain; therefore, no “session” is established. However, the same containment relationships are enforced. Servers and native clients access the domain they are currently in by specifying an empty string, rather than //host:port.

**Note:** When using the Bootstrap object, client and server applications must use the Tobj_Bootstrap::resolve_initial_references() method, not the ORB::resolve_initial_references() method.

### Types of BEA Remote Clients Supported

Table 4-1 shows the types of remote clients that can use the Bootstrap object to access the other environmental objects, such as FactoryFinder, SecurityCurrent, TransactionCurrent, and
interfaceRepository. These clients are provided with the BEA Tuxedo CORBA software. Third-party client ORBs should use the CORBA Interoperable Naming Service.

### Table 4-1  BEA Remote Clients Supported

<table>
<thead>
<tr>
<th>Client</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CORBA C++</td>
<td>CORBA C++ client applications use the BEA Tuxedo C++ environmental objects to access the CORBA objects in a BEA Tuxedo domain, and the BEA Tuxedo Object Request Broker (ORB) to process from CORBA objects. Use the BEA Tuxedo system development commands to build these client applications (see the BEA Tuxedo Command Reference).</td>
</tr>
</tbody>
</table>

### Capabilities and Limitations

Bootstrap objects have the following capabilities and limitations:

- Multiple Bootstrap objects can coexist in a client application, although only one Bootstrap object can own the Current objects (Transaction and Security) at one time. Client applications must invoke `destroy_current()` on the Bootstrap object associated with one domain before obtaining the Current objects on another domain. Although it is possible to have multiple Bootstrap objects that establish connections to different BEA Tuxedo domains, only one set of Current objects is valid. Attempts to obtain other Current objects without destroying the existing Current objects fail.

- Method invocations to any BEA Tuxedo domain that has security enabled other than the domain that provides the valid SecurityCurrent object will fail and return a `CORBA::NO_PERMISSION` exception.

- Method invocations to any BEA Tuxedo domain other than the domain that provides the valid TransactionCurrent object do not execute within the scope of a transaction.

- The transaction and security objects returned by the Bootstrap objects are BEA implementations of the Current objects. If other (“native”) Current objects are present in the environment, they are ignored.

### Bootstrap Object API

The Bootstrap object application programming interface (API) is described first in terms of the OMG Interface Definition Language (IDL) (for portability), and then in C++. The C++ descriptions add the necessary constructor to build a Bootstrap object for a particular BEA Tuxedo domain.
**Tobj Module**

Table 4-2 shows the object reference that is returned for each type ID.

### Table 4-2 Returned Object References

<table>
<thead>
<tr>
<th>ID</th>
<th>Returned Object Reference for C++ Clients</th>
</tr>
</thead>
<tbody>
<tr>
<td>FactoryFinder</td>
<td>FactoryFinder object (Tobj::FactoryFinder)</td>
</tr>
<tr>
<td>InterfaceRepository</td>
<td>InterfaceRepository object (CORBA::Repository)</td>
</tr>
<tr>
<td>NameService</td>
<td>CORBA Naming Service (Tobj::NameService)</td>
</tr>
<tr>
<td>NotificationService</td>
<td>EventChannelFactory object (CosNotifyChannelAdmin::EventChannelFactory)</td>
</tr>
<tr>
<td>SecurityCurrent</td>
<td>SecurityCurrent object (SecurityLevel2::Current)</td>
</tr>
<tr>
<td>TransactionCurrent</td>
<td>OTS Current object (Tobj::TransactionCurrent)</td>
</tr>
<tr>
<td>Tobj_SimpleEventsService</td>
<td>BEA Simple Events ChannelFactory object (Tobj_SimpleEvents::ChannelFactory)</td>
</tr>
</tbody>
</table>

Table 4-3 describes the Tobj module exceptions.

### Table 4-3 Tobj Module Exceptions

<table>
<thead>
<tr>
<th>C++ Exception</th>
<th>Java Exception</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tobj::InvalidName</td>
<td>com.beasys.Tobj::InvalidName</td>
<td>Raised if id is not one of the names specified in Table 4-2. On the server, resolve_initial_references also raises InvalidName when SecurityCurrent is passed.</td>
</tr>
<tr>
<td>Tobj::InvalidDomain</td>
<td>com.beasys.Tobj::InvalidDomain</td>
<td>On the server application, raised if the BEA Tuxedo server environment is not booted.</td>
</tr>
<tr>
<td>CORBA::NO_PERMISSION</td>
<td>org.omg.CORBA::NO_PERMISSION</td>
<td>Raised if id is TransactionCurrent or SecurityCurrent and another Bootstrap object in the client owns the Current objects.</td>
</tr>
</tbody>
</table>
### C++ Mapping

Listing 4-1 shows the C++ declarations in the `Tobj_bootstrap.h` file.

#### Listing 4-1  Tobj_bootstrap.h Declarations

```c++
#include <CORBA.h>

class Tobj_Bootstrap {
public:
  Tobj_Bootstrap(CORBA::ORB_ptr orb, const char* address);
  CORBA::Object_ptr resolve_initial_references(const char* id);
  void register_callback_port(CORBA::Object_ptr objref);
  void destroy_current();
};
```

### Java Mapping

Listing 4-2 shows the `Tobj_Bootstrap.java` mapping.

#### Listing 4-2  Tobj_Bootstrap.java Mapping

```java
package com.beasys;

public class Tobj_Bootstrap {
  public Tobj_Bootstrap(org.omg.CORBA.ORB orb,
                        String address)
```
public class Tobj_Bootstrap {
    public Tobj_Bootstrap(org.omg.CORBA.ORB orb, String address,
                           java.applet.Applet applet)
        throws org.omg.CORBA.SystemException;
    public void register_callback_port(orb.omg.CORBA.Object objref)
        throws org.omg.CORBA.SystemException;
    public org.omg.CORBA.Object
        resolve_initial_references(String id)
        throws Tobj.InvalidName,
               org.omg.CORBA.SystemException;
    public void destroy_current()
        throws org.omg.CORBA.SystemException;
}

Automation Mapping

Listing 4-3 shows Automation Bootstrap interface mapping.

Listing 4-3  Automation (Dual) Bootstrap Interface Mapping

interface DITobj_Bootstrap : IDispatch
{
    HRESULT Initialize(
        [in] BSTR address);
    HRESULT CreateObject(
        [in] BSTR progid,
        [out, retval] IDispatch** rtn);
    HRESULT destroy_current();
};

C++ Member Functions

This section describes the C++ member functions supported by the BEA bootstrapping mechanism.
**Tobj_Bootstrap**

**Synopsis**

The Bootstrap object constructor.

**C++ Mapping**

```cpp
Tobj_Bootstrap(CORBA::ORB_ptr orb, const char* address);
    throws Tobj::BAD_PARAM
            org.omg.CORBA.SystemException;
```

**Parameters**

- **orb**
  
  A pointer to the ORB object in the client. The Bootstrap object uses the `string_to_object` method of `orb` internally.

- **address**
  
  The address of the BEA Tuxedo domain IIOP Listener/Handler.

  **Note:** Multiple `Tobj_Bootstraps` going to the same domain is not supported.

  The address is specified differently depending on the type of client and the level of security required. There can be three types of clients, as follows:

  - **Remote client**

    For a description of the remote clients supported by BEA Tuxedo CORBA, see the section Types of BEA Remote Clients Supported.

    For remote clients, `address` specifies the network address of an IIOP Listener/Handler through which client applications gain access to a BEA Tuxedo domain.

    The address may be specified in either of the following formats:

    - “//hostname:port_number”
    - “#.#.#.#:port_number”
    - “corbaloc://hostname:port_number”
    - “corbalocs://hostname:port_number”

    In the first format, the domain finds an address for `hostname` using the local name resolution facilities (usually DNS). The hostname must be the remote machine, and the local name resolution facilities must unambiguously resolve `hostname` to the address of the remote machine.

    **Note:** The `hostname` must begin with a letter character.
In the second format, the #.#.#.# is in dotted decimal format. In dotted decimal format, each # should be a number from 0 to 255. This dotted decimal number represents the IP address of the remote machine.

In both the first and second formats, port_number is the TCP port number at which the domain process listens for incoming requests. The port_number should be a number between 0 and 65535.

You can specify one or more TCP/IP addresses. You specify multiple addresses using a comma-separated list. For example:

```
//m1.acme:3050
//m1.acme:3050,//m2.acme:3050,//m3.acme:3051
```

If you specify multiple addresses, the BEA Tuxedo software tries the addresses in order, left to right, until a connection is established. If a syntax error is detected in any of the addresses as it is being tried, a BAD_PARAM exception is returned to the caller immediately and the BEA Tuxedo software aborts the attempt to make a connection. For example, if the first address in the comma-separated list shown above were //m1.3050, a syntax error would be detected and the attempt to make a connection would be aborted. If the BEA Tuxedo software encounters the end of the address list before it tries an address that is invalid, that is, a connection cannot be made to any of the addresses listed, the INVALID_DOMAIN exception is returned to the caller. If an exception other than INVALID_DOMAIN is raised, it is returned to the caller immediately.

BEA Tuxedo also supports random address selection. To use random address selection, you can specify any member of an address list as a grouping of pipe-separated (|) network addresses enclosed in parentheses. For example:

```
(//m1.acme:3050|//m2.acme:3050),//m1.acme:7000
```

When you use this format, the BEA Tuxedo system randomly selects one of the addresses enclosed in parentheses, either //m1.acme:3050 or //m2.acme:3050. If an exception other than INVALID_DOMAIN is raised, it is returned to the caller immediately. If a connection cannot be made to the address selected, the next element that follows the addresses enclosed in parentheses is attempted. If the end of the string is encountered before a connection can be made, the INVALID_DOMAIN exception is thrown to the caller.

**Note:** If you specify an address list in the following format:

```
(/m1.acme:3050|/m2.acme:3050),/r1.acme:7000
```

the NULL address in the pipe-separated list is considered invalid. If the BEA Tuxedo software randomly selects the invalid address, the BAD_PARAM exception is returned to the caller and the BEA Tuxedo software aborts the connection attempt.
The address string can be specified either in the `TOBJADDR` environment variable or in the address parameter of the `Tobj_Bootstrap` constructor.

For information about the `TOBJADDR` environment variable, see the section Managing Remote Client Applications in the *Setting Up a BEA Tuxedo Application*. However, the address specified in `Tobj_Bootstrap` always take precedence over the `TOBJADDR` environment variable. To use the `TOBJADDR` environment variable to specify an address string, you must specify an empty string in the `Tobj_Bootstrap` address parameter.

**Note:** For C++ applications, `TOBJADDR` is an environment variable; for Java applications, it is a property; for Java applets, it is an HTML parameter.

The third and fourth formats are called Uniform Resource Locator (URL) address formats and were introduced in the BEA WebLogic Enterprise version 5.1 release. As with the NULL scheme URL address format (`//hostname:port_number`), you use the URL address formats to specify the location of the IIOP Listener/Handler. However, when the `corbaloc` URL address format is used, the client application’s initial connection to the IIOP Listener/Handler is deferred until authentication of the principal’s, or client’s, identity or the first user initiated operation. Using the `corbalocs` URL address format has the same effect on the deferred connection time as `corbaloc`, but, additionally, the client application makes its initial connection to the ISL/ISH using the Secure Sockets Layer (SSL) protocol. Table 4-4 highlights the differences between the two URL address formats.

<table>
<thead>
<tr>
<th>URL Address Formats</th>
<th>Differences in Mode of Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>corbaloc</td>
<td>Invocations to the IIOP Listener/Handler are unprotected. Configuring the IIOP Listener/Handler for the SSL protocol is optional.</td>
</tr>
<tr>
<td></td>
<td><strong>Note:</strong> A principal can secure the bootstrapping process by using the <code>SecurityLevel2::Current::authenticate()</code> operation to specify that certificate-based authentication is to be used.</td>
</tr>
<tr>
<td>corbalocs</td>
<td>Invocations to the IIOP Listener/Handler are protected and the IIOP Listener/Handler or the server ORB must be configured to enable the use of the SSL protocol.</td>
</tr>
</tbody>
</table>

These URL address formats are a subset of the definition of object URLs adopted by the OMG as part of the Interoperable Naming Service submission. The BEA Tuxedo software also extends the URL format described in the OMG Interoperable Naming...
Service submission to support a secure form that is modeled after the URL for secure HTTP, as well as to support the randomize functionality that was added in the BEA WebLogic Enterprise version 4.2.

The `corbaloc` and `corbalocs` URL schemes provide locations that are easily manipulated in both TCP/IP and DNS centric environments. These URL schemes contain a DNS-style `hostname` or IP address and a `port_number`. The following are some examples of the URL formats:

```
corbalocs://host1:1024,(host2:1022|host3:1999)
```

As an enhancement to the URL syntax described in the OMG Interoperable Naming Service submission, the BEA WebLogic Enterprise version 5.1 software extended the syntax to support a list of multiple URLs, each with a different scheme. The following are some examples of the extension:

```
corbalocs://curly:1024,corbaloc://larry:1111,
corbalocs://ctxobj:3434,mthd:3434,corbaloc://force:1111
```

In the above example, if the parser reaches the URL `corbaloc://force:1111`, it resets its internal state as if it had never attempted secure connections and then begins attempting unprotected connections.

**Caution:** Do not mix the use of NULL scheme URL addresses (`//hostname:port_number`) with `corbaloc` and `corbalocs` URL addresses.

**Note:** The Bootstrap object supplied for use with the Netscape embedded Java ORB and JavaSoft JDK ORB does not support `corbaloc` and `corbalocs` URLs.

**Note:** For more information on using the `corbaloc` and `corbalocs` URL address formats, see *Using Security in CORBA Applications*.

**Note:** The network address that is specified in the Bootstrap constructor or in `TOBJADDR` must exactly match the network address in the server application's `UBBCONFIG` file, both the address as well as the capitalization. If the addresses do not match, the invocation to the Bootstrap constructor will fail with the following seemingly unrelated error message:

```
ERROR: Unofficial connection from client at <tcp/ip address>/<port-number>
```

For example, if the network address is specified (using the NULL URL address format) as `//TRIXIE:3500` in the ISL command-line option string in the server
application's UBBCONFIG file, specifying either //192.12.4.6:3500 or //trixie:3500 in the Bootstrap constructor or in TOBJADDR will cause the connection attempt to fail. On UNIX systems, use the `uname -n` command on the host system to determine the capitalization used. On Windows systems, see the host system's network settings in the Control Panel to determine the correct capitalization.

**Note:** The error in the previous note is deferred when the URL address format is used, that is, the error does not occur at the time of Bootstrap object construction because the connection to the ISL/ISH is deferred until later.

- **Native client**
  
  For a native client, the `address` parameter in the Tobj_Bootstrap constructor must always be an empty string (not a NULL pointer). The native client connects to the application that is specified in the TUXCONFIG environment variable. The constructor raises `CORBA::BAD_PARAM` if the address is not empty.

- **Server acting as a client**
  
  When servers need access to the Bootstrap object, they should obtain a reference to it using the TP framework by invoking `TP.bootstrap()`. Servers should not attempt to create a new instance of the Bootstrap object.

**applet (Applies to Java method only)**

This is a pointer to the client applet. If the client applet does not explicitly pass the ISH host and port to the Bootstrap constructor, you can pass this argument, which causes the Bootstrap object to search for the TOBJADDR definition in the HTML file for the applet.

**Exception**

`BAD_PARAM`

Raised if the object is nil or if the host contained in the object does not match the connection or the host address (`//hostname:port_number`) is not in a valid format.

**Description**

A C++ member function (or Java method) that creates Bootstrap objects.

**Return Values**

A pointer to a newly created Bootstrap object.
**Tobj_Bootstrap::register_callback_port**

**Synopsis**

Registers the joint client/server’s listening port in IIOP Handler (ISH).

**C++ Mapping**

```cpp
void register_callback_port(CORBA::Object_ptr objref);
```

**Parameter**

- `objref`
  The object reference created by the joint client/server.

**Exceptions**

- **BAD_PARAM**
  Raised if the object is nil or if the host contained in the object does not match the connection.

- **IMP_LIMIT**
  Raised if the `register_callback_port` method is called more than once.

**Description**

This C++ member function (or Java method) is called to notify the ISH of a listening port in the joint client/server. This method should only be used for joint client/server ORBs that do not support GIOP 1.2 bidirectional capabilities (that is GIOP 1.0 and 1.1 client ORBs). For GIOP 1.0 and 1.1, the ISH supports only one listening port per joint client/server; therefore, the `register_callback_port` method should only be called once per connected joint client/server.

**Usage Notes**

The following information must be given consideration when using this method:

- If the `register_callback_port` method is not invoked by the joint client/server, the callback port is not registered with the ISH and the server defaults to Asymmetric Outbound IIOP. In this case, you must start the server’s IIOP Listener (ISL) with the `-O` option. The `-O` option enables Asymmetric outbound IIOP; otherwise, server-to-client invocations will not be allowed by the ISL/ISH.

- If you are using the BEAWrapper Callbacks API instead of the POA and you want to use bidirectional behavior, you always need to invoke the `register_callback_port` method, even when you are using a ISH that supports GIOP 1.2.
If you want to use bidirectional capability for a callback object, you must invoke the
register_callback_port method before you pass the callback object reference to the
server.

Return Values

None.

Tobj_Bootstrap::resolve_initial_references

Synopsis

Acquires CORBA object references.

C++ Mapping

```c++
CORBA::Object_ptr resolve_initial_references(
    const char* id);
throws Tobj::InvalidName,
    org.omg.CORBA.SystemException;
```

Parameter

`id`

This parameter must be one of the following:

```
"FactoryFinder"
"InterfaceRepository"
"NameService"
"NotificationService"
"SecurityCurrent"
"TransactionCurrent"
"Tobj_SimpleEventsService"
```

Exceptions

InvalidName

Raised if `id` is not one of the names specified above. On the server,
resolve_initial_references also raises Tobj::InvalidName when SecurityCurrent is passed.

CORBA::NO_PERMISSION

Raised if `id` is TransactionCurrent or SecurityCurrent and another Bootstrap object in the
client owns the Current objects.
Description

This C++ member function (or Java method) acquires CORBA object references for the FactoryFinder, SecurityCurrent, TransactionCurrent, NotificationService, Tobj_SimpleEventsService, and InterfaceRepository objects. For the specific object reference, invoke the _narrow function. For example, for FactoryFinder, invoke Tobj::FactoryFinder::_narrow.

Return Values

Table 4-2 shows the object reference that is returned for each type id.

Tobj_Bootstrap::destroy_current()

Synopsis

Destroys the Current objects for the domain represented by the Bootstrap object.

C++ Mapping

void destroy_current();

Exception

Raises CORBA::NO_PERMISSION if the Bootstrap object is not the owner of the Current objects.

Description

This C++ member function invalidates the Current objects for the domain represented by the Bootstrap object. After invoking the destroy_current() method, the Current objects are marked as invalid. Any subsequent attempt to use the old Current objects will throw the exception CORBA::BAD_INV_ORDER. Good programming practice is to release all Current objects before invoking destroy_current().

Note: The destroy_current() method must be invoked on the Bootstrap object for the domain that currently owns the two Current objects (Transaction and Security). This also results in an implicit invocation to logoff for security and implicitly rolls back any transaction that was begun by the client.

The application must invoke destroy_current() before invoking resolve_initial_references for TransactionCurrent or SecurityCurrent on another domain; otherwise, resolve_initial_references raises CORBA::NO_PERMISSION.
Return Values
None.

Java Methods
The Java BEA bootstrapping API supports the following methods:

- Tobj_Bootstrap
- Tobj_Bootstrap.register_callback_port
- Tobj_Bootstrap.resolve_initial_references
- Tobj_Bootstrap.destroy_current
- Tobj_Bootstrap.GetTransactions
- Tobj_Bootstrap.getUserTransaction
- Tobj_Bootstrap.getNativeProperties
- Tobj_Bootstrap.getRemoteProperties

Automation Methods
This section describes the Automation methods supported by the BEA bootstrapping mechanism.

Initialize

Synopsis
Initializes the Bootstrap object into a BEA Tuxedo domain.

MIDL Mapping
HRESULT Initialize(
    [in] BSTR host);

Automation Mapping
    Sub Initialize(address As String)
Parameter

address

The host name and port of the BEA Tuxedo domain IIOP Listener/Handler. One or more TCP/IP addresses can be specified. Multiple addresses are specified using a comma-separated list, as in the C++ mappings. If no address is specified, the value of the TOBJADDR environmental variable is used.

**Note:** The network address that is specified in the Bootstrap constructor or in TOBJADDR must exactly match the network address in the application’s UBBCONFIG file, both the format of the address as well as the capitalization. If the addresses do not match, the invocation to the Bootstrap constructor will fail with the following seemingly unrelated error message:

```
ERROR: Unofficial connection from client at <tcp/ip address>/<port-number>
```

For example, if the network address is specified as `/TRIXIE:3500` in the ISL command-line option string, specifying either `/192.12.4.6:3500` or `/trixie:3500` in the Bootstrap constructor or in TOBJADDR will cause the connection attempt to fail. On UNIX systems, use the `uname -n` command on the host system to determine the capitalization used. On Windows systems, see the host system’s network settings in the Control Panel to determine the correct capitalization.

Return Values

None.

Exceptions

Table 4-5 describes the exceptions.

<table>
<thead>
<tr>
<th>HRESULT</th>
<th>Description</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>ITF_E_NO_PERMISSION</td>
<td>Bootstrap already initialized</td>
<td>The Bootstrap object has already been initialized. To connect to a new BEA Tuxedo domain, you must create a new Bootstrap object.</td>
</tr>
<tr>
<td>E_INVALIDARG</td>
<td>Invalid address parameter</td>
<td>The address supplied is not valid.</td>
</tr>
</tbody>
</table>
CreateObject

Synopsis

Creates an instance of a Current environmental object.

MIDL Mapping

```c
HRESULT CreateObject(
    [in] BSTR progid,
    [out, retval] IDispatch** rtrn);
```

Automation Mapping

```vbnet
Function CreateObject(progid As String) As Object
```

Parameter

- `progid`
  The `progid` of the environmental object to create. Valid `progids` are:
  - Tobj.FactoryFinder
  - Tobj.SecurityCurrent
  - Tobj.TransactionCurrent

Return Value

A reference to the interface pointer of the created environmental object.

Table 4-5  Initialize Exceptions

<table>
<thead>
<tr>
<th>HRESULT</th>
<th>Description</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>E_OUTOFMEMORY</td>
<td>Memory allocation failed</td>
<td>The required memory could not be allocated.</td>
</tr>
<tr>
<td>E_FAIL</td>
<td>Invalid domain</td>
<td>Unable to communicate with the BEA Tuxedo domain at the address specified or TOBJADDR is not defined.</td>
</tr>
<tr>
<td>&lt;SYSTEM ERROR&gt;</td>
<td>Unable to obtain initial object</td>
<td>Unable to initialize the Bootstrap object. The system error causing the failure is returned in the &quot;Number&quot; member of the error object.</td>
</tr>
</tbody>
</table>
Exceptions

Table 4-6 describes the exceptions.

<table>
<thead>
<tr>
<th>Exception</th>
<th>Description</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>ITF_E_NO_PERMISSION_YES</td>
<td>Bootstrap object must be initialized</td>
<td>The Bootstrap object has not been initialized.</td>
</tr>
<tr>
<td>ITF_E_NO_PERMISSION_NO</td>
<td>No permission</td>
<td>If the progid specifies a transaction or security current and another Bootstrap object in the client owns the current objects.</td>
</tr>
<tr>
<td>E_INVALIDARG</td>
<td>Invalid progid parameter</td>
<td>The progid specified is not valid.</td>
</tr>
<tr>
<td>E_INVALIDARG</td>
<td>Invalid name</td>
<td>The requested progid is not one of the valid parameter values specified above.</td>
</tr>
<tr>
<td>E_INVALIDARG</td>
<td>Unknown object</td>
<td>The requested progid is not registered on your system.</td>
</tr>
<tr>
<td>&lt;SYSTEM ERROR&gt; CoCreate Instance() failed</td>
<td></td>
<td>The Bootstrap object could not create an instance of the requested object. The system error is returned in the &quot;Number&quot; member of the error object.</td>
</tr>
</tbody>
</table>

DestroyCurrent

Synopsis

Logs out of the BEA Tuxedo domain and invalidates the TransactionCurrent and SecurityCurrent objects.

MIDL Mapping

```c
HRESULT destroy_current();
```

Automation Mapping

```c
Sub destroy_current()
```
Parameters
None.

Return Value
None.

Exceptions
None.

Bootstrap Object Programming Examples

This section provides the following programming examples that use Bootstrap objects.

– Visual Basic Client Example: Using the Bootstrap Object

Visual Basic Client Example: Using the Bootstrap Object

Listing 4-5 shows how to program a Visual Basic client to use the Bootstrap object.

Listing 4-5  Programming a Client in Visual Basic

'Declare the Bootstrap object
Public oBootstrap As DITobj_Bootstrap

' Declare the FactoryFinder object
Public oBsFactoryFinder As DITobj_FactoryFinder

'Declare factory for Registrar object
Public oRegistrarFactory As DIUniversityB_RegistrarFactory

'Declare actual Registrar object
Public oRegistrarFactory As DIUniversityB_RegistrarFactory

....

'Create the Bootstrap object
Set oBootstrap = CreateObject("Tobj.Bootstrap")

'Connect to the BEA Tuxedo Domain
oBootstrap.Initialize "//host:port"
Interoperable Naming Service Bootstrapping Mechanism

This topic includes the following topics:

- Introduction
- INS Object References
- INS Command-line Options
- INS Object URL Schemes
- Getting a FactoryFinder Object Reference Using INS
- Getting a PrincipalAuthenticator Object Reference Using INS
- Getting a TransactionFactory Object Reference Using INS

Introduction

As of release 8.0, the BEA Tuxedo ORB supports the CORBA Naming Service bootstrapping mechanism (referred to in this document as the Interoperable Naming Service), as specified in Chapters 4 and 13 of the CORBA Specification revision 2.4.2.

This support enables ORBs that implement the Interoperable Naming Service (INS) bootstrapping mechanism to query the BEA Tuxedo server-side ORB to get object references to initial objects such as FactoryFinder and to PrincipalAuthenticator to the BEA Tuxedo environment. This support along with client support for interoperable initial object references enables clients to use the INS bootstrapping mechanism instead of the BEA bootstrapping mechanism.
Note: The CORBA C++ client provided with BEA Tuxedo software may use the INS bootstrapping mechanism, however, for performance reasons, this is not recommended.

**INS Object References**

Table 4-7 shows the object reference that is returned for each type ID.

<table>
<thead>
<tr>
<th>ID</th>
<th>Returned Object Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>FactoryFinder</td>
<td>FactoryFinder object (CORBA::FactoryFinder)</td>
</tr>
<tr>
<td>InterfaceRepository</td>
<td>InterfaceRepository object (CORBA::Repository)</td>
</tr>
<tr>
<td>NameService</td>
<td>CORBA Naming Service object (CORBA::NameService)</td>
</tr>
<tr>
<td>NotificationService</td>
<td>EventChannelFactory object (CosNotifyChannelAdmin::EventChannelFactory)</td>
</tr>
<tr>
<td>POA Current</td>
<td>POACurrent object (CORBA::POACurrent)</td>
</tr>
<tr>
<td>PrincipalAuthenticator</td>
<td>PrincipalAuthenticator object (SecurityLevel2::PrincipalAuthenticator)</td>
</tr>
<tr>
<td>RootPOA</td>
<td>RootPOA object (CORBA::RootPOA)</td>
</tr>
<tr>
<td>Tobj_SimpleEventsService</td>
<td>BEA Simple Events ChannelFactory object (Tobj_SimpleEvents::ChannelFactory)</td>
</tr>
</tbody>
</table>

**INS Command-line Options**

As of release 8.0, BEA Tuxedo CORBA supports the -ORBInitRef and -ORBDefaultInitRef command-line options. For a complete description of these options, see “ORB Initialization Member Function” on page 14-85.

The following example assumes a BEA Tuxedo CORBA IIOP client is talking to a BEA Tuxedo CORBA IIOP server environment:

```plaintext
client_app -ORBid BEA_IIOP -ORBInitRef
            FactoryFinder=corbaloc::myhost:2468/FactoryFinder
```

Given this example, a call to ORB::resolve_initial_references for the FactoryFinder will result in an interoperable initial reference request being sent to the ISL/ISH on myhost at
port 2468. Note that the case of *myhost* must exactly match the case of the host specified for the ISL/ISH in the *tuxconfig* file.

### INS Initialization Operations

To use the INS bootstrapping mechanism, applications programmers must observe the following requirements:

- **BEA Tuxedo CORBA IIOP clients** that want to use the INS initial reference mechanism must now call `ORB::resolve_initial_references` function, instead of the `Tobj_Bootstrap::resolve_initial_references` function. For a syntactical description of `ORB::resolve_initial_references`, see “CORBA::ORB::resolve_initial_references” on page 14-79.

  **Note:** The `Tobj_Bootstrap` API is still supported and its behavior has not changed.

- **BEA Tuxedo CORBA IIOP clients** using the INS initial reference mechanism should use the `ORB::list_initial_services` function instead of the `Tobj_Bootstrap::list_initial_services` function. For a syntactical description of `ORB::list_initial_services`, see “CORBA::ORB::list_initial_services” on page 14-75.

### INS Object URL Schemes

As of release 8.0, BEA Tuxedo CORBA supports an additional Uniform Resource Locator (URL) format to be used for the specification of the location for access to a BEA Tuxedo CORBA server environment and from where to retrieve references to initial object. The new URL format both follows and extends the definition of object URLs adopted by the OMG as part of the INS specification. The URL format described in the INS specification has also been extended to support a secure form modeled after the URL for secure HTTP, as well as the ability to support the randomize functionality initially provided in BEA WebLogic Enterprise version 5.1.

The CORBA 2.4.2 specification requires that three object URL schemes must be supported by a compliant ORB. These schemes are defined as IOR, corbaloc, and corbaname.

**Note:** The new URL string formats may also be passed to the `ORB::string_to_object` function.

#### IOR URL Scheme

The IOR scheme takes the form of a string that is formatted as `IOR:hex_octets`. The scheme name is IOR and the text after the ‘:’ is defined in the CORBA specification. The IOR URL...
scheme is robust and insulates the client from the encapsulated transport information and object key used to reference the object.

**corbaloc URL Scheme**

It is difficult for humans to exchange IORs through nonelectronic means because of their lengths and the text encoding of binary information. The corbaloc and corbalocs URL schemes provide stringified object references in a format that is familiar to people and similar to the popular URL schemes of FTP and HTTP. The URL schemes defined for corbaloc and corbalocs are easily manipulated in both TCP/IP and DNS centric environments. The corbaloc and corbalocs URL contains:

- DNS-style host name or IP address and port
- The version of the IIOP protocol to be used (optional)
- An object key (optional)

By default, corbaloc URLs denote objects that can be contacted over IIOP, while corbalocs URLs denote objects that can be contacted using IIOP over SSL.

Table 4-8 lists the BNF syntax for each URLs element.

<table>
<thead>
<tr>
<th>Table 4-8 BNF Format for URL Elements</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>URL Element</strong></td>
</tr>
</tbody>
</table>
| <corbaloc> | = “corbaloc::”<obj_addr_list>[/”<key_string>]
[,<corbaloc>|<corbalocs>] |
| <corbalocs> | = “corbalocs::”<obj_addr_list>[/”<key_string>]
[,<corbaloc>|<corbalocs>] |
| <obj_addr_list> | = [<obj_addr> ”,”]* <obj_addr> |
| <obj_addr> | = <iiop_prot_addr> | <future_prot_addr> |
| <iiop_prot_addr> | = <iiop_id><iiop_addr> |
| <iiop_id> | = “//” | <iiop_prot_token>”:” |
| <iiop_prot_token> | = “iiop” |
| <iiop_addr> | = [<version> <host> [“:” <port>]] |
| <host> | = DNS-style Host Name | ip_address |
Table 4-8  BNF Format for URL Elements (Continued)

<table>
<thead>
<tr>
<th>URL Element</th>
<th>BNF Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;version&gt;</td>
<td>= &lt;major&gt; “.” &lt;minor&gt; “@”</td>
</tr>
<tr>
<td>&lt;port&gt;</td>
<td>= number</td>
</tr>
<tr>
<td>&lt;major&gt;</td>
<td>= number</td>
</tr>
<tr>
<td>&lt;minor&gt;</td>
<td>= number</td>
</tr>
<tr>
<td>&lt;key_string&gt;</td>
<td>= &lt;string&gt;</td>
</tr>
</tbody>
</table>

Table 4-9 describes each URL element.

Table 4-9  Descriptions of URL Elements

<table>
<thead>
<tr>
<th>URL Element</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>obj_addr_list</td>
<td>A comma-separated list of protocol ID, version, and address information. This list is used in an implementation-defined manner to address the object. An object may be contacted by any of the addresses and protocols. If a failure occurs using the element, the next element in the comma-separated list will be used.</td>
</tr>
<tr>
<td>obj_addr</td>
<td>A protocol identifier, version tag, and a protocol specific address. The right-brace “{”, left-brace “}”, vertical bar “</td>
</tr>
<tr>
<td>iiop_prot_addr</td>
<td>An IIOP protocol identifier, version tag, and address containing a DNS-style host name or IP address.</td>
</tr>
<tr>
<td>iiop_id</td>
<td>Tokens recognized to indicate an IIOP protocol corbaloc.</td>
</tr>
<tr>
<td>iiop_prot_token</td>
<td>An IIOP protocol token, “iiop”.</td>
</tr>
<tr>
<td>iiop_addr</td>
<td>A single address element.</td>
</tr>
<tr>
<td>host</td>
<td>A DNS-style host name or IP address. If not present, the local host is assumed.</td>
</tr>
<tr>
<td>version</td>
<td>A major and minor version number, separated by “.” and followed by “@”. If the version is absent, 1.0 is assumed.</td>
</tr>
<tr>
<td>ip_address</td>
<td>A numeric IP address (dotted decimal notation).</td>
</tr>
</tbody>
</table>
The following are some examples of using the new URL format:

- `corbaloc::555xyz.com:1024,555backup.com:1022,555last.com:1999`
- `corbalocs::555xyz.com:1024,{555backup.com:1022|555last.com:1999}`
- `corbaloc::1.2@555xyz.com:1111`
- `corbalocs::ctxobj:3434,mthd:3434,corbaloc::force:1111`

As an enhancement to the URL syntax described in the INS submission, BEA Tuxedo 8.0 or later has extended the syntax to support a list of multiple URLs, each with a different scheme. The following are some examples of the extension:

- `corbalocs::555xyz.com:1024,corbaloc::1.2@555xyz.com:1111`
- `corbalocs::ctxobj:3434,mthd:3434,corbaloc::force:1111`

In the above example, if the parser reaches the URL `corbaloc::force.com:1111`, it will reset its internal state as if it had never attempted secure connections and then begins attempting unprotected connections.

---

Table 4-9  Descriptions of URL Elements

<table>
<thead>
<tr>
<th>URL Element</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>port</td>
<td>The port number an IIOP Listener/Handler or an initialization agent is listening on. The default is 9999.</td>
</tr>
</tbody>
</table>
| key_string  | A stringified object key that is not NULL-terminated. The key_string uses the escape conventions described in RFC 2396 to map away from octet values that cannot directly be part of a URL. US-ASCII alphanumeric characters are not escaped. Characters outside this range are escaped, except for the following:

- `'` | `@` | `&` | `=` | `+` | `$` |
- `,` | `-` | `_` | `.` | `!` | `~` | `*` |

The key_string corresponds to the octet sequence in the object_key member of a GIOP Request or LocateRequest header as defined in the CORBA specification. |
| string_name | A stringified name with URL escapes as defined in the Internet Engineering Task Force (IETF) RFC 2396. These escape rules insure that URLs can be transferred via a variety of transports without undergoing changes. US-ASCII alphanumeric characters are not escaped. Characters outside this range are escaped, except for the following:

- `'` | `@` | `&` | `=` | `+` | `$` |
- `,` | `-` | `_` | `.` | `!` | `~` | `*` |

The string_name corresponds to the octet sequence in the string_name member of a GIOP Request or LocateRequest header as defined in the CORBA specification.
corbaname URL Scheme

The corbaname URL scheme extends the capabilities of the corbaloc scheme to allow URLs to denote entries in a Naming Service. Resolving corbaname URLs does not require a Naming Service implementation in the ORB core. An example of a corbaname URL is:

corbaname:555objs.com#a/string/path/to/obj

This URL specifies that at host 555objs.com, an object of type NamingContext (with an object key of NamingService) can be found, or alternatively, that an agent running at that location will return a reference to a NamingContext. The stringified name a/string/path/to/obj is then used as the argument to the resolve operation on that NamingContext.

A corbaname URL is similar to a corbaloc URL except that a corbaname URL also contains a stringified name that identifies a binding in a naming context. The # character denotes the start of the stringified name.

The BNF syntax for the URL is listed in Table 4-10.

<table>
<thead>
<tr>
<th>URL Element</th>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;corbaname&gt;</td>
<td>&quot;corbaname:&quot;&lt;corbaloc_obj&gt;[ #&lt;string_name&gt;]</td>
<td>corbaloc_obj is a portion of a corbaname URL that identifies the naming context. The syntax is identical to its use in a corbaloc URL.</td>
</tr>
<tr>
<td>&lt;corbaloc_obj&gt;</td>
<td>&lt;obj_addr_list&gt;[/&lt;key_string&gt;]</td>
<td>For a description of obj_addr_list, see Table 4-9.</td>
</tr>
<tr>
<td>&lt;obj_addr_list&gt;</td>
<td>As defined in a corbaloc URL</td>
<td>For a description of obj_addr_list, see Table 4-9.</td>
</tr>
<tr>
<td>&lt;key_string&gt;</td>
<td>As defined in a corbaloc URL</td>
<td>For a description of key_string, see Table 4-9.</td>
</tr>
<tr>
<td>&lt;string_name&gt;</td>
<td>Stringified Name</td>
<td>empty string</td>
</tr>
</tbody>
</table>

Resolution of a corbaname URL is implemented as a simple extension to corbaloc URL processing. To illustrate the implementation, we will use the following corbaname URL:

corbaname:<corbaloc_obj>[#<string_name>]
The resolution process is as follows:

1. Construct a corbaloc URL of the form corbaloc::<corbaloc_obj> from the corbaname URL.
2. Convert the corbaloc URL to a naming context object reference by calling CORBA::ORB::string_to_object to obtain a CosNaming::NamingContext object.
3. Convert <string_name> to a CosNaming::Name.
4. Invoke the resolve operation on the CosNaming::NamingContext, passing the CosNaming::Name constructed.
5. The object reference returned from CosNaming::NamingContext::resolve should be returned to the caller.

By following this resolution process, you eliminate the possibility of returning an object reference for a naming context that does not exist in the Naming Service. One side effect of this approach is that it requires that stubs for the Naming Service be part of the ORB core or that there be an internal mechanism for sending the request for the resolve operation. Because of the complexity, it is recommended that stubs for the Naming Service be embedded within the ORB core.

**Getting a FactoryFinder Object Reference Using INS**

Listing 4-6 shows an example of how a client application, using INS, gets an object reference to the FactoryFinder object. For a complete code example, see the client application in the University Sample.

**Listing 4-5  Code Example for Getting the FactoryFinder Object**

```c++
// utility to get the registrar
static UniversityW::Registrar_ptr get_registrar(CORBA::ORB_ptr orb)
{

  // Get the factory finder from the ORB:
  CORBA::Object_var v_fact_finder_oref =
    orb->resolve_initial_references("FactoryFinder");

  // Do something with the object reference...
}
```
// Narrow the factory finder:
Tobj::FactoryFinder_var v_fact_finder_ref =
    Tobj::FactoryFinder::_narrow(v_fact_finder_oref.in());

// Use the factory finder to find the
// university's registrar factory:
CORBA::Object_var v_reg_fact_oref =
    v_fact_finder_ref->find_one_factory_by_id(
        UniversityW::tc_RegistrarFactory->id());

// Narrow the registrar factory:
UniversityW::RegistrarFactory_var v_reg_fact_ref =
    UniversityW::RegistrarFactory::_narrow(
        v_reg_fact_oref.in());

// Return the university's registrar:
return v_reg_fact_ref->find_registrar();

---

Getting a PrincipalAuthenticator Object Reference Using INS

Listing 4-7 shows an example of how a client application, using INS, gets an object reference to the PrincipalAuthenticator object. For a complete code example, see the client application in the University Sample.

Listing 4-6  Code Example for Getting the PrincipalAuthenticator Object

// utility to log on to the security system
static SecurityLevel2::PrincipalAuthenticator_ptr logon(
    CORBA::ORB_ptr orb,
    const char* program_name,
    UniversityW::StudentId stu_id) {
}
// Get a Principal Authenticator directly from the ORB:
CORBA::Object_var v_pa_obj = 
    orb->resolve_initial_references("PrincipalAuthenticator");

// Narrow the Principal Authenticator :
SecurityLevel2::PrincipalAuthenticator_var v_pa = 
    SecurityLevel2::PrincipalAuthenticator::_narrow( 
        v_pa_obj.in());

---

Getting a TransactionFactory Object Reference Using INS

As of release 8.0, BEA Tuxedo CORBA supports the use of the CORBA Transaction Service Interface for beginning transactions. Using the 
ORB::resolve_initial_references("FactoryFinder") function, a client gets an object reference to a FactoryFinder. The client then uses the FactoryFinder to get a reference to a TransactionFactory, that it in turn uses to create (begin) a transaction.

Listing 4-8 shows an example of how a client application, using INS, gets an object reference to the TransactionFactory object. For a complete code example, see the client application in the University Sample.

Listing 4-7   Code Example for a Client Application That Uses INS

// Get the factory finder from the ORB:
CORBA::Object_var v_fact_finder_oref = 
    orb->resolve_initial_references("FactoryFinder");

// Narrow the factory finder :
Tobj::FactoryFinder_var v_fact_finder_ref = 
    Tobj::FactoryFinder::_narrow(v_fact_finder_oref.in());

// Get the TransactionFactory from the FactoryFinder
CORBA::Object_var v_txn_fac_oref = 
    v_fact_finder_ref->find_one_factory_by_id( 
        "IDL:omg.org/CosTransactions/TransactionFactory:1.0");

// Narrow the TransactionFactory object reference
CosTransactions::TransactionFactory_var v_txn_fac_ref =

// Get a Principal Authenticator directly from the ORB:
CORBA::Object_var v_pa_obj = 
    orb->resolve_initial_references("PrincipalAuthenticator");

// Narrow the Principal Authenticator :
SecurityLevel2::PrincipalAuthenticator_var v_pa = 
    SecurityLevel2::PrincipalAuthenticator::_narrow( 
        v_pa_obj.in());
The sequence of events using the INS bootstrapping mechanism is as follows:

1. Use `ORB::resolve_initial_references` to get a FactoryFinder.
2. Use the FactoryFinder to get a TransactionFactory.
3. Use the create operation on TransactionFactory to begin a transaction.
4. From the Control object returned from the create operation, use the `get_terminator` method to get the transaction terminator interface.
5. Use the commit or rollback operation on the terminator to end or abort the transaction.

The TransactionFactory returns objects that adhere to the standard CORBA Transaction Service interfaces instead of the BEA delegated interfaces. This means that a third party ORB can use their ORB’s `resolve_initial_references` function to get a reference to a TransactionFactory from a BEA Tuxedo CORBA server and use stubs generated from standard OMG IDL to act on the instances returned.

**Restrictions**

For the BEA Tuxedo 8.0 release or later, the actions of the TransactionFactory and the client’s Current are not coordinated. This means that clients should use one mechanism or the other to control and get status about transactions, not both. Also, only the interfaces and operations listed in Table 4-11 are supported. The other operations, as described in the OMG IDL, return the `CORBA::NO_IMPLEMENT` exception.

<table>
<thead>
<tr>
<th>Table 4-11 Supported INS Interfaces and Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interface</td>
</tr>
<tr>
<td>-------------------</td>
</tr>
<tr>
<td>TransactionFactory</td>
</tr>
<tr>
<td>Control</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Interface</td>
</tr>
<tr>
<td>--------------</td>
</tr>
<tr>
<td>Terminator</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Coordinator</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>
The FactoryFinder interface provides clients with one object reference that serves as the single point of entry into the BEA Tuxedo domain. The BEA Tuxedo NameManager provides the mapping of factory names to object references for the FactoryFinder. Multiple FactoryFinders and NameManagers together provide increased availability and reliability. In this release the level of functionality has been extended to support multiple domains.

**Note:** The NameManager is not a naming service, such as CORBA services Naming Service, but is merely a vehicle for storing registered factories.

In the BEA Tuxedo environment, application factory objects are used to create objects that clients interact with to perform their business operations (for example, TellerFactory and Teller). Application factories are generally created during server initialization and are accessed by both remote clients and clients located within the server application.

The FactoryFinder interface and the NameManager services are contained in separate (nonapplication) servers. A set of application programming interfaces (APIs) is provided so that both client and server applications can access and update the factory information.

The support for multiple domains in this release benefits customers that need to scale to a large number of machines or who want to partition their application environment. To support multiple domains, the mechanism used to find factories in a BEA Tuxedo environment has been enhanced to allow factories in one domain to be visible in another. The visibility of factories in other domains is under the control of the system administrator.
Capabilities, Limitations, and Requirements

During server application initialization, application factories need to be registered with the NameManager. Clients can then be provided with the object reference of a FactoryFinder to allow them to retrieve a factory object reference based on associated names that were created when the factory was registered.

The following functional capabilities, limitations, and requirements apply to this release:

- The FactoryFinder interface is in compliance with the `CosLifeCycle::FactoryFinder` interface.
- Server applications can register and unregister application factories with the CORBA services Naming Service.
- Clients can access objects using a single point of entry—the FactoryFinder.
- Clients can construct names for objects using a simplified BEA scheme made possible by BEA Tuxedo extensions to the CORBA services interface or the more general CORBA scheme.
- Multiple FactoryFinder and NameManager services can be used to increase availability and reliability in the event that one FactoryFinder or NameManager should fail.
- Support for multiple domains. Factories in one domain can be configured to be visible in another domain under administrative control.
- Two NameManager services, at a minimum, must be configured, preferably on different machines, to maintain the factory-to-object reference mapping across process failures. If both NameManagers fail, the master NameManager, which has been keeping a persistent journal of the registered factories, recovers the previous state by processing the journal so as to re-establish its internal state.
- One NameManager must be designated as the Master and the Master NameManager must be started before the Slave. If the master NameManager is started after one or more Slaves, the Master assumes that it is in recovery mode instead of in initializing mode.

Functional Description

The BEA Tuxedo CORBA environment promotes the use of the factory design pattern as the primary means for a client to obtain a reference to an object. Through the use of this design pattern, client applications require a mechanism to obtain a reference to an object that acts as a factory for another object. Because the BEA Tuxedo environment has chosen CORBA as its
visible programming model, the mechanism used to locate factories is modeled after the
FactoryFinder as described in the CORBA services Specification, Chapter 6 “Life Cycle Service,”

In the CORBA FactoryFinder model, application servers register active factories with a
FactoryFinder. When an application server’s factory becomes inactive, the application server
removes the corresponding registration from the FactoryFinder. Client applications locate
factories by querying a FactoryFinder. The client application can control the references to the
factory object returned by specifying criteria that is used to select one or more references.

Locating a FactoryFinder

A client application must obtain a reference to a FactoryFinder before it can begin locating an
appropriate factory. To obtain a reference to a FactoryFinder in the domain to which a client
application is associated, the client application can use either of two bootstrapping mechanisms:

- **Invoke the** Tobj_Bootstrap::resolve_initial_references **operation with a value of
  “FactoryFinder”. This operation returns a reference to a FactoryFinder that is in the
domain to which the client application is currently attached. You should use this
mechanism if you are using the BEA Tuxedo client software. For more information, see
the section Tobj_Bootstrap::resolve_initial_references.**

- **Invoke the** CORBA::ORB::resolve_initial_references **operation with a value of
  “FactoryFinder”. This operation returns a reference to a FactoryFinder that is in the
domain to which the client application is currently attached. You should use this
mechanism if you are using a third-party client ORB. For more information, see the section
CORBA::ORB::resolve_initial_references.**

**Note:** The references to the FactoryFinder that are returned to the client application can be
references to factory objects that are registered on the same machine as the
FactoryFinder, on a different machine than the FactoryFinder, or possibly in a different
domain than the FactoryFinder.

Registering a Factory

For a client application to be able to obtain a reference to a factory, an application server must
register a reference to any factory object for which it provides an implementation with the
FactoryFinder (see Figure 5-1). Using the BEA Tuxedo CORBA TP Framework, the registration
of the reference for the factory object can be accomplished using the TP::register_factory
operation, once a reference to a factory object has been created. The reference to the factory
object, along with a value that identifies the factory, is passed to this operation. The registration
of references to factory objects is typically done as part of initialization of the application (normally as part of the implementation of the operation `Server::initialize`).

**Figure 5-1 Registering a Factory Object**

When the server application is shutting down, it must unregister any references to factory objects that it has previously registered in the application server. This is done by passing the same reference to the factory object, along with the corresponding value used to identify the factory, to the `TP::unregister_factory` operation. Once unregistered, the reference to the factory object can then be destroyed. The process of unregistering a factory with the FactoryFinder is typically done as part of the implementation of the `Server::release` operation. For more information about these operations, see the section Server Interface.

**C++ Mapping**

Listing 5-1 shows the C++ class (static) methods. For more information about these methods, see the sections `TP::register_factory()` and `TP::unregister_factory()`.

**Listing 5-1 C++ Mappings for the Factory Registration Pseudo OMG IDL**

```cpp
#include <TP.h>
static void TP::register_factory(
    CORBA::Object_ptr factory_or, const char* factory_id);
```
static void TP::unregister_factory(
    CORBA::Object_ptr factory_or, const char* factory_id);

The TP.h header file contains the two method declarations. This file must be included in any server application that wants to use these methods.

A server application generally includes this header file within the application file that contains the methods for application server initialization and release.

**Locating a Factory**

For a client application to request a factory to create a reference to an object, it must first obtain a reference to the factory object. The reference to the factory object is obtained by querying a FactoryFinder with specific selection criteria (see Figure 5-2). The criteria are determined by the format of the particular FactoryFinder interface and method used.

**Figure 5-2 Locating a Factory Object**

BEA Tuxedo CORBA extends the CosLifeCycle::FactoryFinder interface by introducing four methods in addition to the find_factories() method declared for the FactoryFinder. Therefore, using the Tobj extensions, a client can use either the find_factories() or find_factories_by_id() methods to obtain a list of application factories. A client can also
use the `find_one_factory()` or `find_one_factory_by_id()` method to obtain a single application factory, and `list_factories()` to obtain a list of all registered factories.

**Note:** You can use the BEA Tuxedo CORBA extensions to the `CosLifeCycle::FactoryFinder` interface if you use the `Tobj_Bootstrap` object, however, use of the `Tobj_Bootstrap` object is not required to locate a factory. If you use CORBA INS, you can use the `find_factories()` method provided by the `CosLifeCycle::FactoryFinder` interface.

The `CosLifeCycle::FactoryFinder` interface defines a `factory_key`, which is a sequence of `id` and `kind` strings conforming to the `CosNaming Name` shown below. The `kind` field of the `NameComponent` for all application factories is set to the string `FactoryInterface` by the TP Framework when an application factory is registered. Applications supply their own value for the `id` field.

Assuming that the CORBA services Life Cycle Service modules are contained in their own file (`ns.idl` and `lcs.idl`, respectively), only the OMG IDL code for that subset of both files that is relevant for using the BEA Tuxedo FactoryFinder is shown in the following listings.

### CORBA services Naming Service Module OMG IDL

Listing 5-2 shows the portions of the `ns.idl` file that are relevant to the FactoryFinder.

#### Listing 5-2  CORBA services Naming OMG IDL

```idl
// ------  ns.idl  ------
module CosNaming {  
    typedef string Istring;  
    struct NameComponent {  
        Istring id;  
        Istring kind;  
    };  
    typedef sequence <NameComponent> Name;  
};

```
CORBAservices Life Cycle Service Module OMG IDL

Listing 5-3 shows the portions of the `lcs.idl` file that are relevant to the FactoryFinder.

```
// -----  lcs.idl  -----
#include "ns.idl"
module CosLifeCycle{
  typedef CosNaming::Name Key;
  typedef Object Factory;
  typedef sequence<Factory> Factories;
  exception NoFactory{ Key search_key; }
  interface FactoryFinder {
    Factories find_factories(in Key factory_key) raises(NoFactory);
  };
};
```

Tobj Module OMG IDL

Listing 5-4 shows the Tobj Module OMG IDL.

```
// -----  Tobj.idl  -----
module Tobj {
  // Constants
  const string FACTORY_KIND = "FactoryInterface";
};
```
// Exceptions
exception CannotProceed { }; 
exception InvalidDomain { }; 
exception InvalidName { }; 
exception RegistrarNotAvailable { }; 

// Extension to LifeCycle Service
struct FactoryComponent {
    CosLifeCycle::Key factory_key;
    CosLifeCycle::Factory factory_ior;
};

typedef sequence<FactoryComponent> FactoryListing;

interface FactoryFinder : CosLifeCycle::FactoryFinder {
    CosLifeCycle::Factory find_one_factory(in CosLifeCycle::Key factory_key)
        raises (CosLifeCycle::NoFactory,
               CannotProceed,
               RegistrarNotAvailable);
    CosLifeCycle::Factory find_one_factory_by_id(in string factory_id)
        raises (CosLifeCycle::NoFactory,
               CannotProceed,
               RegistrarNotAvailable);
    CosLifeCycle::Factories find_factories_by_id(in string factory_id)
        raises (CosLifeCycle::NoFactory,
               CannotProceed,
               RegistrarNotAvailable);
    FactoryListing list_factories()
        raises (CannotProceed,
               RegistrarNotAvailable);
};
Locating Factories in Another Domain

Typically, a FactoryFinder returns references to factory objects that are in the same domain as the FactoryFinder itself. However, it is possible to return references to factory objects in domains other than the domain in which a FactoryFinder exists. This can occur if a FactoryFinder contains information about factories that are resident in another domain (see Figure 5-3). A FactoryFinder finds out about these interdomain factory objects through configuration information that describes the location of these other factory objects.

When a FactoryFinder receives a request to locate a factory object, it must first determine if a reference to a factory object that meets the specified criteria exists. If there is registration information for a factory object that matches the criteria, the FactoryFinder must then determine if the factory object is local to the current domain or needs to be imported from another domain. If the factory object is from the local domain, the FactoryFinder returns the reference to the factory object to the client.

If, on the other hand, the information indicates that the actual factory object is from another domain, the FactoryFinder delegates the request to an interdomain FactoryFinder in the appropriate domain. As a result, only a FactoryFinder in the same domain as the factory object will contain an actual reference to the factory object. The interdomain FactoryFinder is responsible for returning the reference of the factory object to the local FactoryFinder, which subsequently returns it to the client.

Figure 5-3 Inter-Domain FactoryFinder Interaction
Why Use BEA Tuxedo CORBA Extensions?

The BEA Tuxedo software extends the interfaces defined in the CORBAservices specification, Chapter 6 “Life Cycle Service,” December 1997, published by the Object Management Group, for the following reasons:

- Although the CORBA-defined approach is powerful and allows various selection criteria, the interface used to query a FactoryFinder can be complicated to use.

- Additionally, if the selection criterion specified by the client application is not specific enough, it is possible that more than one reference to a factory object may be returned. If this occurs, it is not immediately obvious what a client application should do next.

- Finally, the CORBAservices specification did not specify a standardized mechanism through which an application server is to register a factory object.

Therefore, BEA Tuxedo extends the interfaces defined in the CORBAservices specification to make using a FactoryFinder easier. The extensions are manifested as refined interfaces to the FactoryFinder that are derived from the interfaces specified in the CORBAservices specification.

Creating Application Factory Keys

Two of the five methods provided by the FactoryFinder interface accept `CosLifeCycle::Keys`, which corresponds to `CosNaming::Name`. A client must be able to construct these keys.

The CosNaming Specification describes two interfaces that constitute a Names Library interface that can be used to create and manipulate `CosLifeCycle::Keys`. The pseudo OMG IDL statements for these interfaces is described in the following section.

Names Library Interface Pseudo OMG IDL

**Note:** This information is taken from the *CORBA* services: Common Object Services Specification, pp. 3-14 to 18. Revised Edition: March 31, 1995. Updated: November 1997. Used with permission by OMG.

To allow the representation of names to evolve without affecting existing client applications, it is desirable to hide the representation of names from the client application. Ideally, names themselves would be objects; however, names must be lightweight entities that are efficient to create, manipulate, and transmit. As such, names are presented to programs through the names library.

The names library implements names as pseudo-objects. A client application makes calls on a pseudo-object in the same way it makes calls on an ordinary object. Library names are described
in pseudo-IDL (to suggest the appropriate language binding). C++ client applications use the same client language bindings for pseudo-IDL (PIDL) as they use for IDL.

Pseudo-object references cannot be passed across OMG IDL interfaces. As described in Chapter 3 of the CORBA services: Common Object Services Specification, in the section “The CosNaming Module,” the CORBA services Naming Service supports the NamingContext OMG IDL interface. The names library supports an operation to convert a library name into a value that can be passed to the name service through the NamingContext interface.

**Note:** It is not a requirement to use the names library in order to use the CORBA services Naming Service.

The names library consists of two pseudo-IDL interfaces, the LNameComponent interface and the LName interface, as shown in Listing 5-5.

**Listing 5-5  Names Library Interfaces in Pseudo-IDL**

```idl
interface LNameComponent { // PIDL
   const short MAX_LNAME_STRLEN = 128;
   exception NotSet{ };
   exception OverFlow{ };
   string get_id
      raises (NotSet);
   void set_id(in string i)
      raises (OverFlow);
   string get_kind()
      raises(NotSet);
   void set_kind(in string k)
      raises (OverFlow);
   void destroy();
};
interface LName {// PIDL
   exception NoComponent{ };
   exception OverFlow{ };
   exception InvalidName{ };
   LName insert_component(in unsigned long i,
      in LNameComponent n)
      raises (NoComponent, OverFlow);
```

Functional Description
Creating a Library Name Component
To create a library name component pseudo-object, use the following C/C++ function:

```c
LNameComponent create_lname_component();    // C/C++
```

The returned pseudo-object can then be operated on using the operations shown in Listing 5-5.

Creating a Library Name
To create a library name pseudo-object, use the following C/C++ function:

```c
LName create_lname();        // C/C++
```

The returned pseudo-object reference can then be operated on using the operations shown in Listing 5-5.

The LNameComponent Interface
A name component consists of two attributes: identifier and kind. The LNameComponent interface defines the operations associated with these attributes, as follows:

```c
string get_id() raises (NotSet);
void set_id(in string k);
string get_kind()
```
raises(NotSet);
void set_kind(in string k);

get_id
   The get_id operation returns the identifier attribute’s value. If the attribute has not
   been set, the NotSet exception is raised.

set_id
   The set_id operation sets the identifier attribute to the string argument.

get_kind
   The get_kind operation returns the kind attribute’s value. If the attribute has not been
   set, the NotSet exception is raised.

set_kind
   The set_kind operation sets the kind attribute to the string argument.

The LName Interface
The following operations are described in this section:

- Destroying a library name component pseudo-object
- Inserting a name component
- Getting the i\textsuperscript{th} name component
- Deleting a name component
- Number of name components
- Testing for equality
- Testing for order
- Producing an OMG IDL form
- Translating an OMG IDL form
- Destroying a library name pseudo-object

Destroying a Library Name Component Pseudo-Object
The destroy operation destroys library name component pseudo-objects.

void destroy();
Inserting a Name Component

A name has one or more components. Each component except the last is used to identify names of subcontexts. (The last component denotes the bound object.) The `insert_component` operation inserts a component after position \( i \).

\[
\text{LName insert_component}(\text{in unsigned long } i, \text{ in LNameComponent lnc}) \text{ raises(NoComponent, OverFlow)};
\]

If component \( i-1 \) is undefined and component \( i \) is greater than 1 (one), the `insert_component` operation raises the `NoComponent` exception.

If the library cannot allocate resources for the inserted component, the `OverFlow` exception is raised.

Getting the \( i \)-th Name Component

The `get_component` operation returns the \( i \)-th component. The first component is numbered 1 (one).

\[
\text{LNameComponent get_component}(\text{in unsigned long } i) \text{ raises(NoComponent)};
\]

If the component does not exist, the `NoComponent` exception is raised.

Deleting a Name Component

The `delete_component` operation removes and returns the \( i \)-th component.

\[
\text{LNameComponent delete_component}(\text{in unsigned long } i) \text{ raises(NoComponent)};
\]

If the component does not exist, the `NoComponent` exception is raised.

After a `delete_component` operation has been performed, the compound name has one fewer component and components previously identified as \( i+1 \ldots n \) are now identified as \( i \ldots n-1 \).

Number of Name Components

The `num_components` operation returns the number of components in a library name.

\[
\text{unsigned long num_components();}
\]

Testing for Equality

The `equal` operation tests for equality with library name \( \text{ln} \).
boolean equal(in LName ln);

Testing for Order

The less_than operation tests for the order of a library name in relation to library name ln.

boolean less_than(in LName ln);

This operation returns TRUE if the library name is less than the library name ln passed as an argument. The library implementation defines the ordering on names.

Producing an OMG IDL Form

Pseudo-objects cannot be passed across OMG IDL interfaces. The library name is a pseudo-object; therefore, it cannot be passed across the OMG IDL interface for the CORBAservices Naming Service. Several operations in the NamingContext interface have arguments of an OMG IDL-defined structure, Name. The following PIDL operation on library names produces a structure that can be passed across the OMG IDL request.

Name to_idl_form()
    raises(InvalidName);

If the name is of length 0 (zero), the InvalidName exception is returned.

Translating an IDL Form

Pseudo-objects cannot be passed across OMG IDL interfaces. The library name is a pseudo-object; therefore, it cannot be passed across the OMG IDL interface for the CORBAservices Naming Service. The NamingContext interface defines operations that return an IDL struct of type Name. The following PIDL operation on library names sets the components and kind attribute for a library name from a returned OMG IDL defined structure, Name.

void from_idl_form(in Name n);

Destroying a Library Name Pseudo-Object

The destroy operation destroys library name pseudo-objects.

void destroy();

C++ Mapping

The Names Library pseudo OMG IDL interface maps to the C++ classes shown in Listing 5-6, which can be found in the NamesLib.h header file.
Two BEA Tuxedo extensions to CORBA are included to support scalability. Specifically, the
LNameComponent::set_id() and LNameComponent::set_kind() methods raise an
Overflow exception if the length of the input string exceeds MAX_LNAME_STRLEN. This length
coincides with the maximum length of the BEA Tuxedo object ID (OID) and interface name. For
a detailed description of the Library Name class, see the section Names Library Interface Pseudo
OMG IDL.

Listing 5-6  Library Name Class

const short MAX_LNAME_STRLEN = 128;

class LNameComponent {
public:
  class NotSet{ };  
  class OverFlow{ };
  static LNameComponent* create_lname_component();
  void destroy();
  const char* get_id() const throw (NotSet);
  void set_id(const char* i) throw (OverFlow);
  const char* get_kind() const throw (NotSet);
  void set_kind(const char* k) throw (OverFlow);
};

class LName {
public:
  class NoComponent{ };  
  class OverFlow{ };  
  class InvalidName{ };
  static LName* create_lname();
  void destroy();
  LName* insert_component(const unsigned long i,
                           LNameComponent* n)
                     throw (NoComponent, OverFlow);
  const LNameComponent* get_component(
                               const unsigned long i) const
                     throw (NoComponent);
  const LNameComponent* delete_component(
                               const unsigned long i)
throw (NoComponent); 
unsigned long num_components() const;
CORBA::Boolean equal(const LName* ln) const;
CORBA::Boolean less_than(
    const LName* ln) const; // not implemented
CosNaming::Name* to_idl_form()
    throw (InvalidName);
void from_idl_form(const CosNaming::Name& n);

Java Mapping
The Names Library pseudo OMG IDL interface maps to the Java classes contained in the com.beasys.Tobj package, shown in Listing 5-7. All exceptions are contained in the same package.

For a detailed description of the Library Name class, refer to Chapter 3 in the CORBA Services: Common Object Services Specification.

Listing 5-7 Java Mapping for LNameComponent

```java
public class LNameComponent {
    public static LNameComponent create_lname_component();
    public static final short MAX_LNAME_STRING = 128;
    public void destroy();
    public String get_id() throws NotSet;
    public void set_id(String i) throws OverFlow;
    public String get_kind() throws NotSet;
    public void set_kind(String k) throws OverFlow;
};
```

```java
public class LName {
    public static LName create_lname();
    public void destroy();
    public LName insert_component(long i, LNameComponent n)
        throws NoComponent, OverFlow;
    public LNameComponent get_component(long i)
```
C++ Member Functions and Java Methods

This section describes the FactoryFinder C++ member functions and Java methods.

**Note:** All FactoryFinder member functions, except the `less_than` member function in `LName`, are implemented in both C++ and Java.

The following methods are described in this section:

- `CosLifeCycle::FactoryFinder::find_factories`
- `Tobj::Factoryfinder::find_one_factory`
- `Tobj::Factoryfinder::find_one_factory_by_id`
- `Tobj::Factoryfinder::find_factories_by_id`
- `Tobj::Factoryfinder::list_factories`

**Note:** The `CosLifeCycle::FactoryFinder::find_factories` method is the standard CORBA `CosLifeCycle` method. The four Tobj methods are extensions to the `CosLifeCycle` interface and, therefore, inherit the attributes of the `CosLifeCycle` interface.

**CosLifeCycle::FactoryFinder::find_factories**

**Synopsis**

Obtains a sequence of factory object references.
C++ Mapping

```cpp
CosLifeCycle::Factories *
CORBA::Object_ptr CosLifeCycle::FactoryFinder::find_factories(
    const CosNaming::Name& factory_key)
throw (CosLifeCycle::NoFactory);
```

Java Mapping

```java
import org.omg.CosLifeCycle.*;

public org.omg.CORBA.Object[] find_factories(
    org.omg.CosNaming.NameComponent[] factory_key)
throws org.omg.CosLifeCycle.NoFactory;
```

Parameter

`factory_key`

This parameter is an unbounded sequence of NameComponents (tuple of `<id, kind>` pairs) that uniquely identifies a factory object reference. A NameComponent is defined as having two members: an `id` and a `kind`, both of type `string`. The `id` field is used to represent the identity of factory object. The `kind` field is used to indicate how the value of the `id` field should be interpreted. References to factory object registered using the operation `TP::register_factory` will have a kind value of "FactoryInterface".

Exception

- `CORBA::BAD_PARAM`
  Indicates that the value of an input parameter has an inappropriate value or is invalid. Of particular importance, the exception is raised if no value or a NULL value for the parameter `factory_key` is specified.
- `CosLifeCycle::NoFactory`
  Indicates that there are no factories registered that match the information in the `factory_key` parameter.

Description

The `find_factories` method is called by an application to obtain a sequence of factory object references. The operation is passed a key used to identify the desired factory. The key is a name, as defined by the CORBA services Naming service. More than one factory may match the key, and, if that is the case, the FactoryFinder returns a sequence of factories.
The scope of the key is the FactoryFinder. The FactoryFinder assigns no semantics to the key. It simply matches keys. It makes no guarantees about the interface or implementation of the returned factories or objects they create.

Key values are considered equal if they are of equal length (same number of elements in the sequence), and if every NameComponent value in the key matches the corresponding NameComponent value at the exact same location in the key that was specified when the reference to the factory object was registered.

Return Values

An unbounded sequence of references to factory objects that match the information specified as the value of the factory_key parameter. In C++, the method returns a sequence of object references of type CosLifeCycle::Factory. In Java, the method returns an unbounded array of object references of type org.omg.CORBA.Object.

If the operation raises an exception, the return value is invalid and does not need to be released by the caller.

Tobj::FactoryFinder::find_one_factory

Synopsis

Obtains a reference to a single factory object.

C++ Mapping

virtual CosLifeCycle::Factory_ptr
find_one_factory( const CosNaming::Name& factory_key) = 0;

Java Mapping

public org.omg.CORBA.Object
find_one_factory( org.omg.CosNaming.NameComponent[] factory_key)
throws
    org.omg.CosLifeCycle.NoFactory,
    com.beasys.Tobj.CannotProceed,
    com.beasys.Tobj.RegistrarNotAvailable;
Parameter

factory_key
This parameter is an unbounded sequence of NameComponents (tuple of <id, kind> pairs) that uniquely identifies a factory object reference. A NameComponent is defined as having two members: an id and a kind, both of type string. The id field is used to represent the identity of factory object. The kind field is used to indicate how the value of the id field should be interpreted. References to factory object registered using the operation TP::register_factory will have a kind value of “FactoryInterface”.

Exceptions

CORBA::BAD_PARAM
Indicates that the value of an input parameter has an inappropriate value or is invalid. Of particular importance, the exception is raised if no value or a NULL value for the parameter factory_key is specified.

CosLifeCycle::NoFactory
Indicates that there are no factories registered that match the information in the factory_key parameter.

Tobj::CannotProceed
Indicates that the FactoryFinder or NameManager encountered an internal error while attempting to locate a reference for a factory object. Error information is written to the user log.

Tobj::RegistrarNotAvailable
Indicates that the FactoryFinder could not communicate with the NameManager. Error information is written to the user log.

Description

The find_one_factory method is called by an application to obtain a reference to a single factory object whose key matches the value of the key specified as input to the method. If more than one factory object is registered with the specified key, the FactoryFinder selects one factory object based on the FactoryFinder’s load balancing scheme. As a result, invoking the find_one_factory method multiple times using the same key may return different object references.

The scope of the key is the FactoryFinder. The FactoryFinder assigns no semantics to the key. It simply matches keys. It makes no guarantees about the interface or implementation of the returned factory or objects they create.
Key values are considered equal if they are of equal length (same number of elements in the sequence), and if every NameComponent value in the key matches the corresponding NameComponent value at the exact same location in the key that was specified when the reference to the factory object was registered.

Return Values

An object reference for a factory object. In C++, the method returns an object reference of type CosLifeCycle::Factory. In Java, the method returns an object reference of type org.omg.CORBA.Object.

If the operation raises an exception, the return value is invalid and does not need to be released by the caller.

Tobj::FactoryFinder::find_one_factory_by_id

Synopsis

Obtains a reference to a single factory object.

C++ Mapping

virtual CosLifeCycle::Factory_ptr
   find_one_factory_by_id( const char * factory_id) = 0;

Java Mapping

public org.omg.CORBA.Object
   find_one_factory_by_id( java.lang.String factory_id)
   throws
      org.omg.CosLifeCycle.NoFactory,
      com.beasys.Tobj.CannotProceed,
      com.beasys.Tobj.RegistrarNotAvailable;

Parameter

factory_id

A NULL-terminated string that contains a value that is used to identify the registered factory object to be found. The value of the factory_id parameter is used as the value of the id field of a NameComponent that has a kind field with the value “FactoryInterface” when comparing against registered references for factory objects.
Exceptions

CORBA::BAD_PARAM
Indicates that the value of an input parameter has an inappropriate value or is invalid. Of particular importance, the exception is raised if no value or a NULL value for the parameter factory_key is specified.

CosLifeCycle::NoFactory
Indicates that there are no factories registered that match the information in the factory_key parameter.

Tobj::CannotProceed
Indicates that the FactoryFinder or NameManager encountered an internal error while attempting to locate a reference for a factory object. Error information is written to the user log.

Tobj::RegistrarNotAvailable
Indicates that the FactoryFinder could not communicate with the NameManager. Error information is written to the user log.

Description

The find_one_factory_by_id method is called by an application to obtain a reference to a single factory object whose registration ID matches the value of the ID specified as input to the method. If more than one factory object is registered with the specified ID, the FactoryFinder selects one factory object based on the FactoryFinder’s load balancing scheme. As a result, invoking the find_one_factory_by_id operation multiple times using the same ID may return different object references.

The find_one_factory_by_id method behaves the same as the find_one_factory operation that was passed a key that contains a single NameComponent with an id field that contains the same value as the factory_id parameter and a kind field that contains the value “FactoryInterface”.

The registered identifier for a factory is considered equal to the value of the factory_id parameter if the result of constructing a CosLifeCycle::Key structure containing a single NameComponent that has the factory_id parameter as the value of the id field and the value “FactoryInterface” as the value of the kind field. The values must match exactly in all respects (case, location, etc.).
Return Values

An object reference for a factory object. In C++, the method returns an object reference of type
`CosLifeCycle::Factory`. In Java, the method returns an object reference of type
`org.omg.CORBA.Object`.

If the operation raises an exception, the return value is invalid and does not need to be released
by the caller.

**Tobj::FactoryFinder::find_factories_by_id**

Synopsis

Obtains a sequence of one or more factory object references.

C++ Mapping

```cpp
virtual CosLifeCycle::Factories *
    find_factories_by_id( const char * factory_id) = 0;
```

Java Mapping

```java
public org.omg.CORBA.Object[]
    find_factories_by_id( java.lang.String factory_id)
    throws
        org.omg.CosLifeCycle.NoFactory,
        com.beasys.Tobj.CannotProceed,
        com.beasys.TobjRegistrarNotAvailable;
```

Parameter

`factory_id`

A NULL-terminated string that contains a value that is used to identify the registered
factory object to be found.

The value of the `factory_id` parameter is used as the value of the `id` field of a
NameComponent that has a `kind` field with the value “FactoryInterface” when
comparing against registered references for factory objects.

Exceptions

**CORBA::BAD_PARAM**

Indicates that the value of an input parameter has an inappropriate value or is invalid. Of
particular importance, the exception is raised if no value or a NULL value for the
parameter `factory_key` is specified.
CosLifeCycle::NoFactory
   Indicates that there are no factories registered that match the information in the
   factory_key parameter.

Tobj::CannotProceed
   Indicates that the FactoryFinder or NameManager encountered an internal error while
   attempting to locate a reference for a factory object.
   Error information is written to the user log.

Tobj::RegistrarNotAvailable
   Indicates that the FactoryFinder could not communicate with the NameManager.
   Error information is written to the user log.

Description
   The find_factories_by_id method is called by an application to obtain a sequence of one or
   more factory object references. The method is passed a NULL-terminated string that contains the
   identifier of the factory to be located. If more than one factory object is registered with the
   specified ID, the FactoryFinder will return a list of object references for the matching registered
   factory objects.

   The find_factories_by_id method behaves the same as the find_factory operation that
   was passed a key that contains a single NameComponent with an id field that contains the same
   value as the factory_id parameter and a kind field that contains the value
   "FactoryInterface".

   The registered identifier for a factory is considered equal to the value of the factory_id
   parameter if the result of constructing a CosLifeCycle::Key structure containing a single
   NameComponent that has the factory_id parameter as the value of the id field and the value
   "FactoryInterface" as the value of the kind field. The values must match exactly in all
   respects (case, location, etc.).

Return Values
   An unbounded sequence of references to factory objects that match the information specified as
   the value of the factory_key parameter. In C++, the method returns a sequence of object
   references of type CosLifeCycle::Factory. In Java, the method returns an unbounded array of
   object references of type org.omg.CORBA.Object.

   If the operation raises an exception, the return value is invalid and does not need to be released
   by the caller.
Tobj::Factoryfinder::list_factories

Synopsis

Obtains a list of factory objects currently registered with the FactoryFinder.

C++ Mapping

```cpp
virtual FactoryListing * list_factories() = 0;
```

Java Mapping

```java
public com.beasys.Tobj.FactoryComponent[] list_factories()
throws
    com.beasys.Tobj.CannotProceed,
    com.beasys.Tobj.RegistrarNotAvailable;
```

Exception

Tobj::CannotProceed

Indicates that the FactoryFinder or NameManager encountered an internal error while attempting to locate a reference for a factory object.

Error information is written to the user log.

Tobj::RegistrarNotAvailable

Indicates that the FactoryFinder could not communicate with the NameManager.

Error information is written to the user log.

Description

The `list_factories` method is called by an application to obtain a list of the factory objects currently registered with the FactoryFinder. The method returns both the key used to register the factory, as well as a reference to the factory object.

The number of factories returned by `list_factories` will be one more than the ones registered by the user. For example, if the user registered four factories then the number of factories returned by `list_factories` will be five.

**Note:** This change in behavior is because the OMG Transaction Service specification version 1.1 in section 2.1.2 specifies that the Transaction Factory is located using the FactoryFinder interface of the Life Cycle Service. Hence the Transaction factory is registered internally by the product with the FactoryFinder.
Return Values

An unbounded sequence of Tobj::FactoryComponent. Each occurrence of a
Tobj::FactoryComponent in the sequence contains a reference to the registered factory object,
as well as the CosLifeCycle::Key that was used to register that factory object.

If the operation raises an exception, the return value is invalid and does not need to be released
by the caller.

Automation Methods

This section describes the DITobj_FactoryFinder Automation methods.

DITobj_FactoryFinder.find_one_factory

Synopsis

Obtains a single application factory.

MIDL Mapping

HRESULT find_one_factory(
    [in] VARIANT factory_key,
    [in,out,optional] VARIANT* exceptionInfo,
    [out,retval] IDispatch** returnValue);

Automation Mapping

Function find_one_factory(factory_key, [exceptionInfo]) As Object

Parameters

factory_key

This parameter contains a safe array of DICosNaming_NameComponent (<id, kind>
value pairs) that uniquely identifies a factory object reference.

exceptionInfo

An optional input argument that enables the application to get additional exception data if
an error occurred.
Exceptions

NoFactory
This exception is raised if the FactoryFinder cannot find an application factory object reference that corresponds to the input factory_key.

CannotProceed
This exception is raised if the FactoryFinder or CORBA services Naming Service encounter an internal error during the search with the error being written to the user log (ULOG). Notify the operations staff immediately if this exception is raised. Depending on the severity of the internal error, the server running the FactoryFinder or CORBA services Naming Service may have terminated. If a FactoryFinder service has terminated, start a new FactoryFinder service. If a CORBA services Naming Service has terminated and there is another CORBA services Naming Service running, start a new CORBA services Naming Service. If no naming services servers are running, restart the application.

RegistrarNotAvailable
This exception is raised if the FactoryFinder object cannot locate the CORBA services Naming Service object. Notify the operations staff immediately if this exception is raised. If no naming services servers are running, restart the application.

Description
This member function instructs the FactoryFinder to return one application factory object reference whose key matches the input factory_key. To accomplish this, the member function performs an equality match; that is, every NameComponent <id, kind> pair in the input factory_key must exactly match each <id, kind> pair in the application factory’s key. If multiple factory keys contain the input factory_key, the FactoryFinder selects one factory key, based on an internally defined load balancing scheme. Invoking find_one_factory multiple times using the same id may return different object references.

Return Values
Returns a reference to an interface pointer for the application factory.

DITobj_FactoryFinder.find_one_factory_by_id

Synopsis
Obtains a single application factory.
MIDL Mapping

```c
HRESULT find_one_factory_by_id(
    [in] BSTR factory_id,
    [in,out,optional] VARIANT* exceptionInfo,
    [out,retval] IDispatch** returnValue);
```

Automation Mapping

```c
Function find_one_factory_by_id(factory_id As String,
                                [exceptionInfo]) As Object
```

Parameters

- **factory_id**
  This parameter represents a string identifier that is used to identify the kind or type of application factory. For some suggestions as to the composition of this string, see Creating CORBA Server Applications.

- **exceptionInfo**
  An optional input argument that enables the application to get additional exception data if an error occurred.

Exceptions

- **NoFactory**
  This exception is raised if the FactoryFinder cannot find an application factory object reference that corresponds to the input `factory_id`.

- **CannotProceed**
  This exception is raised if the FactoryFinder or CORBAservices Naming Service encounter an internal error during the search, with the error being written to the user log (`ULOG`). Notify the operations staff immediately if this exception is raised. Depending on the severity of the internal error, the server running the FactoryFinder or the CORBAservices Naming Service may have terminated. If a FactoryFinder service has terminated, start a new FactoryFinder service. If a CORBAservices Naming Service has terminated and there is another CORBAservices Naming Service running, start a new CORBAservices Naming Service. If there are no naming services running, restart the application.

- **RegistrarNotAvailable**
  This exception is raised if the FactoryFinder object cannot locate the CORBAservices Naming Service object. Notify the operations staff immediately if this exception is raised. If no naming service servers are running, restart the application.
Description

This member function instructs the FactoryFinder to return one application factory object reference whose id in the key matches the method’s input factory_id. To accomplish this, the member function performs an equality match (that is, the input factory_id must exactly match the id in the <id,kind> pair in the application factory’s key). If multiple factory keys contain the input factory_id, the FactoryFinder selects one factory key, based on an internally defined load balancing scheme. Invoking find_one_factory_by_id multiple times using the same id may return different object references.

Return Values

Returns a reference to an interface pointer for the application factory.

DITobj.FactoryFinder.find_factories_by_id

Synopsis

Obtains a list of application factories.

MIDL Mapping

HRESULT find_factories_by_id(
    [in] BSTR factory_id,
    [in,out,optional] VARIANT* exceptionInfo,
    [out,retval] VARIANT* returnValue);

Automation Mapping

Function find_factories_by_id(factory_id As String,
                             [exceptionInfo])

Parameters

factory_id

This parameter represents a string identifier that will be used to identify the kind or type of application factory. The Creating CORBA Client Applications online document provides some suggestions as to the composition of this string.

exceptionInfo

An optional input argument that enables the application to get additional exception data if an error occurred.
Exceptions

**NoFactory**
This exception is raised if the FactoryFinder cannot find an application factory object reference that corresponds to the input `factory_key` or `factory_id`.

**CannotProceed**
This exception is raised if the FactoryFinder or CORBA services Naming Service encounter an internal error during the search with the error being written to the user log (`ULOG`). Notify the operations staff immediately if this exception is raised. Depending on the severity of the internal error, the server running the FactoryFinder or CORBA services Naming Service may have terminated. If a FactoryFinder service has terminated, start a new FactoryFinder service. If a CORBA services Naming Service has terminated and there is another CORBA services Naming Service running, start a new CORBA services Naming Service. If no naming services servers are running, restart the application.

**RegistrarNotAvailable**
This exception is raised if the FactoryFinder object cannot locate the CORBA services Naming Service object. Notify the operations staff immediately if this exception is raised. If no naming services servers are running, restart the application.

Description

This member function instructs the FactoryFinder to return a list of application factory object references whose `id` in the keys match the method’s input `factory_id`. To accomplish this, the member function performs an equality match (that is, the input `factory_id` must exactly match each `id` in the `<id,kind>` pair in the application factory’s keys).

Return Values

Returns a variant containing an array of interface pointers to application factories.

**DITobj_FactoryFinder.find_factories**

Synopsis

Obtains a list of application factories.

MIDL Mapping

```c
HRESULT find_factories(
    [in] VARIANT factory_key,
    [in,out,optional] VARIANT* exceptionInfo,
    [out,retval] VARIANT* returnValue);
```
Automation Mapping

Function find_factories(factory_key, [exceptionInfo])

Parameters

factory_key
This parameter contains a safe array of DICosNaming_NameComponents (<id, kind> value pairs) that uniquely identifies a factory object reference.

exceptionInfo
An optional input argument that enables the application to get additional exception data if an error occurred.

Exception

NoFactory
This exception is raised if the FactoryFinder cannot find an application factory object reference that corresponds to the input factory_key.

Description

The find_factories method instructs the FactoryFinder to return a list of server application factory object references whose keys match the method's input key. The BEA Tuxedo system assumes that an equality match is to be performed. This means that for the two sequences of <id,kind> pairs (those corresponding to the input key and those in the application factory's keys), each are of equal length; for every pair in one sequence, there is an identical pair in the other.

Return Values

Returns a variant containing an array of interface pointers to application factories.

DITobj_FactoryFinder.list_factories

Synopsis

Lists all of the application factory names and object references.

MIDL Mapping

HRESULT list_factories(
    [in,out,optional] VARIANT* exceptionInfo,
    [out,retval] VARIANT* returnValue);
Automation Mapping

Function list_factories([exceptionInfo])

Parameter

exceptionInfo
An optional input argument that enables the application to get additional exception data if an error occurred.

Exception

CannotProceed
This exception is raised if the FactoryFinder or the CORBA services Naming Service encounter an internal error during the search with the error being written to the user log (ULOG). Notify the operations staff immediately if this exception is raised. Depending on the severity of the internal error, the server running the FactoryFinder or the CORBA services Naming Service may have terminated. If a FactoryFinder service has terminated, start a new FactoryFinder service. If a CORBA services Naming Service has terminated and there is another CORBA services Naming Service running, start a new CORBA services Naming Service. If there are no naming service servers running, restart the application.

RegistrarNotAvailable
This exception is raised if the FactoryFinder object cannot locate the CORBA services Naming Service object. Notify the operations staff immediately if this exception is raised. It is possible that no naming service servers are running. Restart the application.

Description
This method instructs the FactoryFinder to return a list containing all of the factory keys and associated object references for application factories registered with the CORBA services Naming Service.

Return Values
Returns a variant containing an array of DITobj_FactoryComponent objects. The FactoryComponent object consists of a variant containing an array of DICosNaming_NameComponent objects and an interface pointer to the application factory.

Programming Examples
This section describes how to program using the FactoryFinder interface.

Note: Remember to check for exceptions in your code.
Using the FactoryFinder Object

A FactoryFinder object is used by programmers to locate a reference to a factory object. The FactoryFinder object provides operations to obtain one or more references to factory objects based on the criteria specified.

There can be more than one FactoryFinder object in a process address space. Multiple references to a FactoryFinder object must be supported. A FactoryFinder object is semi-stateful in that it maintains state about the association between FactoryFinder objects within a domain and a particular IIOP Server Listener/Handler (ISL/ISH) through which to access the domain.

All FactoryFinder objects support the CosLifeCycle::FactoryFinder interface as defined in CORBA services Specification, Chapter 6 “Life Cycle Service,” December 1997, published by the Object Management Group. The interface contains one operation that is used to obtain one or more references to factory objects that meet the criteria specified.

Registering a Reference to a Factory Object

The following code fragment (Listing 5-8) shows how to use the TP Framework interface to register a reference to a factory object with a FactoryFinder.

Listing 5-8  Server Application: Registering a Factory

// Server Application: Registering a factory.
// C++ Example.
TP::register_factory( factory_obj.in( ), "TellerFactory" );

Obtaining a Reference to a FactoryFinder Object Using the CosLifeCycle::FactoryFinder Interface

The following code fragment (Listing 5-9) shows how to use of the CORBA-compliant interface to obtain one or more references to factory objects.

Listing 5-9  Client Application: Getting a FactoryFinder Object Reference

// Client Application: Obtaining the object reference
// to factory objects.
CosLifeCycle::Key_var  factory_key = new CosLifeCycle::Key( );
factory_key ->length(1);
factory_key[0].id = string_dupalloc( "strlen("TellerFactory") +1");
factory_key[0].kind = string_dupalloc(
  strlen(""FactoryInterface") + 1);
strcpy( factory_key[0].id, ""TellerFactory" ");
strcpy( facory_key[0].kind, ""FactoryInterface" ");
CosLifeCycle::Factories_var *  flp = ff_np -->
  find_factories( factory_key.in( ) );

Obtaining a Reference to a FactoryFinder Object Using the Extensions Bootstrap object

The following code fragment (Listing 5-10) shows how to use of the BEA Tuxedo extensions Bootstrap object to obtain a reference to a FactoryFinder object.

Listing 5-10   Client Application: Finding One Factory Using the Tobj Approach

// Client Application: Finding one factory using the Tobj
// approach.
Tobj_Bootstrap *  bsp = new Tobj_Bootstrap(
  orb_ptr.in( ), host_port );
CORBA::Object_varptr  ff_op = bsp -->
  resolve_initial_references( "FactoryFinder" );
Tobj::FactoryFinder_ptrvar   ff_np =
  Tobj::FactoryFinder::narrow( ff_op );

Note: You can used the BEA Tuxedo CORBA extensions to the
CosLifeCycle::FactoryFinder interface if you use the Tobj_Bootstrap object,
however, use of the Tobj_Bootstrap object is not required to locate a factory. If you use
CORBA INS, you can use the find_factories() method provided by the
CosLifeCycle::FactoryFinder interface.
Using Extensions to the FactoryFinder Object

BEA Tuxedo extends the FactoryFinder object with functionality to support similar capabilities to those provided by the operations defined by CORBA, but with a much simpler and more restrictive signature. The enhanced functionality is provided by defining the Tobj::FactoryFinder interface. The operations defined for the Tobj::FactoryFinder interface are intended to provide a focused, simplified form of the equivalent capability defined by CORBA. An application developer can choose to use the CORBA-defined or BEA Tuxedo extensions when developing an application. The interface Tobj::FactoryFinder is derived from the CosLifeCycle::FactoryFinder interface.


The implementation of the extended FactoryFinder object requires users to supply either a CosLifeCycle::Key, as in the CORBA-defined CosLifeCycle::FactoryFinder interface, or a NULL-terminated string containing the identifier of a factory object to be located.

Obtaining One Factory Using Tobj::FactoryFinder

The following code fragment (Listing 5-11) shows how to use the BEA Tuxedo extensions interface to obtain one reference to a factory object based on an identifier.

Listing 5-11   Client Application: Finding Factories Using the BEA Tuxedo Extensions Approach

```c
CosLifeCycle::Factory_ptrvar  fp_obj = ff_np ->
   find_one_factory_by_id( "TellerFactory" );
```

Obtaining One or More Factories Using Tobj::FactoryFinder

The following code fragment (Listing 5-12) shows how to use the BEA Tuxedo extensions to obtain one or more references to factory objects based on an identifier.

Listing 5-12   Client Application: Finding One or More Factories Using the BEA Tuxedo Extensions Approach

```c
CosLifeCycle::Factories * _var  flp = ff_np ->
   find_factories_by_id( "TellerFactory" );
```
Security Service

For a detailed discussion of Security, see *Using Security in CORBA Applications*. This document provides an introduction to crytography and other concepts associated with the BEA Tuxedo security features, a description of how to secure your BEA Tuxedo applications using the security features, and a guide to the use of the application programming interfaces (APIs) in the Security Service.

A PDF file of *Using Security in CORBA Applications* is also provided in the online documentation.
Transactions Service

For a detailed discussion of Transactions, see *Using CORBA Transactions*. This document provides an introduction to transactions, a description of the application programming interfaces (APIs), and a guide to the use of the application programming interfaces (APIs) to develop applications.

A PDF file of *Using CORBA Transactions* is also provided in the online documentation.
CHAPTER 8

Notification Service

For a detailed discussion of the Notification Service, see Using the CORBA Notification Service. This document provides an introduction to the Notification Service, a description of the application programming interfaces (APIs), and a guide to the use of the APIs to develop applications.

A PDF file of Using the CORBA Notification Service is also provided in the online documentation.
CHAPTER 9

Request-Level Interceptors

For a detailed discussion of request-level interceptors, see *Using CORBA Request-Level Interceptors*. This document provides an introduction to request-level interceptors, a description of the application programming interfaces (APIs), and a guide to the use of the APIs to implement request-level interceptors.

A PDF file of *Using CORBA Request-Level Interceptors* is also provided in the online documentation.
This chapter describes the BEA Tuxedo CORBA Interface Repository interfaces.

**Notes:** Most of the information in this chapter is taken from Chapter 10 of the *Common Object Request Broker: Architecture and Specification*, Revision 2.4.2, February 2001. The OMG information has been modified as required to describe the BEA Tuxedo CORBA implementation of the Interface Repository interfaces. Used with permission of the OMG.

The BEA Tuxedo CORBA Java client and BEA Tuxedo CORBA Java client ORB were deprecated in Tuxedo 8.1 and are no longer supported in Tuxedo 9.x. All BEA Tuxedo CORBA Java client and BEA Tuxedo CORBA Java client ORB text references, associated code samples, etc. should only be used:

- to help implement/run third party Java ORB libraries, and
- for programmer reference only.

Technical support for third party CORBA Java ORBs should be provided by their respective vendors. BEA Tuxedo does not provide any technical support or documentation for third party CORBA Java ORBs.

The BEA Tuxedo CORBA Interface Repository contains the interface descriptions of the CORBA objects that are implemented within the BEA Tuxedo domain.

The Interface Repository is based on the CORBA definition of an Interface Repository. It offers a proper subset of the interfaces defined by CORBA; that is, the APIs that are exposed to programmers are implemented as defined by the *Common Object Request Broker: Architecture and Specification* Revision 2.4. However, not all interfaces are supported. In general, the
interfaces required to read from the Interface Repository are supported, but the interfaces required to write to the Interface Repository are not. Additionally, not all TypeCode interfaces are supported.

Administration of the Interface Repository is done using tools specific to the BEA Tuxedo software. These tools allow the system administrator to create an Interface Repository, populate it with definitions specified in Object Management Group Interface Definition Language (OMG IDL), and then delete interfaces. Additionally, an administrator may need to configure the system to include an Interface Repository server. For a description of the Interface Repository administration commands, see the BEA Tuxedo Command Reference and Setting Up a BEA Tuxedo Application.

Several abstract interfaces are used as base interfaces for other objects in the Interface Repository. A common set of operations is used to locate objects within the Interface Repository. These operations are defined in the abstract interfaces IRObj ect, Container, and Contained described in this chapter. All Interface Repository objects inherit from the IRObj ect interface, which provides an operation for identifying the actual type of the object. Objects that are containers inherit navigation operations from the Container interface. Objects that are contained by other objects inherit navigation operations from the Contained interface. The IDLType interface is inherited by all Interface Repository objects that represent OMG IDL types, including interfaces, typedefs, and anonymous types. The TypedefDef interface is inherited by all named noninterface types.

The IRObj ect, Contained, Container, IDLType, and TypedefDef interfaces are not instantiable.

All string data in the Interface Repository are encoded as defined by the ISO 8859-1 character set.

**Note:** The Write interface is not documented in this chapter because the BEA Tuxedo software supports only read access to the Interface Repository. Any attempt to use the Write interface to the Interface Repository will raise the exception `CORBA::NO_IMPLEMENT`.

## Structure and Usage

The Interface Repository consists of two distinct components: the database and the server. The server performs operations on the database.

The Interface Repository database is created and populated using the `idl2ir` administrative command. For a description of this command, see the BEA Tuxedo Command Reference and Setting Up a BEA Tuxedo Application. From the programmer’s point of view, there is no write access to the Interface Repository. None of the write operations defined by CORBA are supported, nor are set operations on nonread-only attributes.
Read access to the Interface Repository database is always through the Interface Repository server; that is, a client reads from the database by invoking methods that are performed by the server. The read operations as defined by the *CORBA Common Object Request Broker: Architecture and Specification*, Revision 2.4, are described in this chapter.

**Programming Information**

The interface to a server is defined in the OMG IDL file. How the OMG IDL file is accessed depends on the type of client being built. Three types of clients are considered: stub based, Dynamic Invocation Interface (DII).

Client applications that use stub-style invocations need the OMG IDL file at build time. The programmer can use the OMG IDL file to generate stubs, and so forth. (For more information, see *Creating CORBA Client Applications.*) No other access to the Interface Repository is required.

Client applications that use the Dynamic Invocation Interface (DII) need to access the Interface Repository programmatically. The interface to the Interface Repository is defined in this chapter and is discussed in “Building Client Applications” on page 10-4. The exact steps taken to access the Interface Repository depend on whether the client is seeking information about a specific object, or browsing the Interface Repository to find an interface. To obtain information about a specific object, clients use the CORBA,Object::_get_interface method to obtain an InterfaceDef object. (Refer to CORBA::Object::_get_interface for a description of this method.) Using the InterfaceDef object, the client can get complete information about the interface.

Before a DII client can browse the Interface Repository, it needs to obtain the object reference of the Interface Repository to start the search.

DII clients use the Bootstrap object to obtain the object reference. (For a description of this method, see the section Tobj_Bootstrap::register_callback_port.) Once the client has the object reference, it can navigate the Interface Repository, starting at the root.

To obtain a reference to a Interface Repository in the domain to which a client application is associated, the client application can use either of two bootstrapping mechanisms:

- **Invoke the Tobj_Bootstrap::resolve_initial_references operation with a value of “CORBA::Repository”.** This operation returns a reference to a InterfaceRepository object that is in the domain to which the client application is currently attached. You should use this mechanism if you are using the BEA Tuxedo client software. For more information, see the section Tobj_Bootstrap::resolve_initial_references.
* Invoke the CORBA::ORB::resolve_initial_references operation with a value of "CORBA::Repository". This operation returns a reference to a InterfaceRepository object that is in the domain to which the client application is currently attached. You should use this mechanism if you are using a third-party client ORB. For more information, see the section CORBA::ORB::resolve_initial_references.

**Note:** To use the DII, the OMG IDL file must be stored in the Interface Repository.

**Performance Implications**

All run-time access to the Interface Repository is via the Interface Repository server. Because there is considerable overhead in making requests of a remote server application, designers need to be aware of this. For example, consider the interaction required to use an object reference to obtain the necessary information to make a DII invocation on the object reference. The steps are as follows:

1. The client application invokes the _get_interface operation on the CORBA::Object to get the InterfaceDef object associated with the object in question. This causes a message to be sent to the ORB that created the object reference.
2. The ORB returns the InterfaceDef object to the client.
3. The client invokes one or more _is_a operations on the object to determine what type of interface is supported by the object.
4. After the client has identified the interface, it invokes the describe_interface operation on the Interface object to get a full description of the interface (for example, version number, operations, attributes, and parameters). This causes a message to be sent to the Interface Repository, and a reply is returned.
5. The client is now ready to construct a DII request.

**Building Client Applications**

Clients that use the Interface Repository need to link in Interface Repository stubs. How this happens is specific to the vendor. If the client application is using the BEA Tuxedo ORB, the BEA Tuxedo software provides the stubs in the form of a library. Therefore, programmers do not need to use the Interface Repository OMG IDL file to build the stubs. The Interface Repository definitions are contained within the CORBA.h file, but they are not included by default.
Getting Initial References to the InterfaceRepository Object

Note: To use the Interface Repository definitions, you must define the ORB_INCLUDE_REPOSITORY macro before including CORBA.h in your client application code (for example: #Define ORB_INCLUDE_REPOSITORY).

If the client application is using a third-party ORB (for example, ORBIX) the programmer must use the mechanisms that are provided by that vendor. This might include generating stubs from the OMG IDL file using the IDL compiler supplied by the vendor, simply linking against the stubs provided by the vendor, or some other mechanism.

Some third-party ORBs provide a local Interface Repository capability. In this case, the local Interface Repository is provided by the vendor and is populated with the interface definitions that are needed by that client.

Getting Initial References to the InterfaceRepository Object

You use the Bootstrap object to get an initial reference to the InterfaceRepository object. For a description of the Bootstrap object method, see the command Tobj_Bootstrap::resolve_initial_references.

Interface Repository Interfaces

Client applications use the interfaces defined by CORBA to access the Interface Repository. This section contains descriptions of each interface that is implemented in the BEA Tuxedo software.

Note: The BEA Tuxedo CORBA implementation of the Interface Repository only supports the read operations on the interfaces. The write operations are not implemented.

Supporting Type Definitions

Several types are used throughout the Interface Repository interface definitions.

```corba
module CORBA {
    typedef string Identifier;
    typedef string ScopedName;
    typedef string RepositoryId;

    enum DefinitionKind {
        dk_none, dk_all,
        dk_Attribute, dk_Constant, dk_Exception, dk_Interface,
    }
}
```
Identifiers are the simple names that identify modules, interfaces, value types, value members, value boxes, constants, typedefs, exceptions, attributes, operations, and native types. They correspond exactly to OMG IDL identifiers. An Identifier is not necessarily unique within an entire Interface Repository; it is unique only within a particular Repository, ModuleDef, InterfaceDef, ValueDef, or OperationDef.

A ScopedName is a name made up of one or more identifiers separated by double colons (::). They correspond to OMG IDL scoped names. An absolute ScopedName is one that begins with double colons (::) and unambiguously identifies a definition in a Repository. An absolute ScopedName in a Repository corresponds to a global name in an OMG IDL file. A relative ScopedName does not begin with double colons (::) and must be resolved relative to some context.

A RepositoryId is an identifier used to uniquely and globally identify a module, interface, value type, value member, value box, native type, constant, typedef, exception, attribute, or operation. Because RepositoryIds are defined as strings, they can be manipulated (for example, copied and compared) using a language binding’s string manipulation routines.

A DefinitionKind identifies the type of an Interface Repository object.

**IObject Interface**

The base interface IObject (shown below) represents the most generic interface from which all other Interface Repository interfaces are derived, even the Repository itself.

```corba
module CORBA {
    interface IObject {
        readonly attribute DefinitionKind def_kind;
    }
};
```

The def_kind attribute identifies the type of the definition.
Contained Interface

The Contained interface (shown below) is inherited by all Interface Repository interfaces that are contained by other Interface Repository objects. All objects within the Interface Repository, except the root object (Repository) and definitions of anonymous (ArrayDef, StringDef, and SequenceDef), and primitive types are contained by other objects.

```
module CORBA {
    typedef string VersionSpec;

    interface Contained : IRObjec t {
        readonly attribute RepositoryId id;
        readonly attribute Identifier name;
        readonly 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 ();
    };
}
```

An object that is contained by another object has an id attribute that identifies it globally, and a name attribute that identifies it uniquely within the enclosing Container object. It also has a version attribute that distinguishes it from other versioned objects with the same name. The BEA Tuxedo CORBA Interface Repository does not support simultaneous containment or multiple versions of the same named object.

Contained objects also have a defined_in attribute that identifies the Container within which they are defined. Objects can be contained either because they are defined within the containing object (for example, an interface is defined within a module) or because they are inherited by the containing object (for example, an operation may be contained by an interface because the interface inherits the operation from another interface). If an object is contained through inheritance, the defined_in attribute identifies the InterfaceDef or ValueDef from which the object is inherited.

The absolute_name attribute is an absolute ScopedName that identifies a Contained object uniquely within its enclosing Repository. If this object’s defined_in attribute references a
Repository, the absolute_name is formed by concatenating the string “::” and this object’s name attribute. Otherwise, the absolute_name is formed by concatenating the absolute_name attribute of the object referenced by this object’s defined_in attribute, the string “::”, and this object’s name attribute.

The containing_repository attribute identifies the Repository that is eventually reached by recursively following the object’s defined_in attribute.

The within operation returns the list of objects that contain the object. If the object is an interface or module, it can be contained only by the object that defines it. Other objects can be contained by the objects that define them and by the objects that inherit them.

The describe operation returns a structure containing information about the interface. The description structure associated with each interface is provided below with the interface’s definition. The kind of definition described by the structure returned is provided with the returned structure. For example, if the describe operation is invoked on an attribute object, the kind field contains dk_Attribute and the value field contains an any, which contains the AttributeDescription structure.

## Container Interface

The base interface Container is used to form a containment hierarchy in the Interface Repository. A Container can contain any number of objects derived from the Contained interface. All Containers, except for Repository, are also derived from Contained.

```corba
module CORBA {
    typedef sequence <Contained> ContainedSeq;

    interface Container : IRObjec{ {
        Contained lookup (in ScopedName search_name);

        ContainedSeq contents ( {
            in DefinitionKind limit_type,
            in boolean exclude_inherited
        });

        ContainedSeq lookup_name ( {
            in Identifier search_name,
            in long levels_to_search,
            in 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
};

The lookup operation locates a definition relative to this container, given a scoped name using the OMG IDL rules for name scoping. An absolute scoped name (beginning double colons (::)) locates the definition relative to the enclosing Repository. If no object is found, a nil object reference is returned.

The contents operation returns the list of objects directly contained by or inherited into the object. The operation is used to navigate through the hierarchy of objects. Starting with the Repository object, a client uses this operation to list all of the objects contained by the Repository, all of the objects contained by the modules within the Repository, all of the interfaces and value types within a specific module, and so on.

limit_type
If limit_type is set to dk_all, objects of all types are returned. For example, if this is an InterfaceDef, the attribute, operation, and exception objects are all returned. If limit_type is set to a specific interface, only objects of that type are returned. For example, only attribute objects are returned if limit_type is set to dk_Attribute.

exclude_inherited
If set to TRUE, inherited objects (if there are any) are not returned. If set to FALSE, all contained objects (whether contained due to inheritance or because they were defined within the object) are returned.
The `lookup_name` operation is used to locate an object by name within a particular object or within the objects contained by that object. The `describe_contents` operation combines the `contents` operation and the `describe` operation. For each object returned by the contents operation, the description of the object is returned (that is, the object’s `describe` operation is invoked and the results are returned).

The `lookup_name` operation is used to locate an object by name within a particular object or within the objects contained by that object.

`search_name`

Specifies which name is to be searched for.

`levels_to_search`

Controls whether the lookup is constrained to the object the operation is invoked on, or whether the lookup should search through objects contained by the object as well. Setting `levels_to_search` to -1 searches the current object and all contained objects. Setting `levels_to_search` to 1 searches only the current object. Use of values of `levels_to_search` of 0 or of negative numbers other than -1 is undefined.

The `describe_contents` operation combines the contents operation and the `describe` operation. For each object returned by the contents operation, the description of the object is returned (i.e., the object’s `describe` operation is invoked and the results returned).

`max_returned_objs`

Limits the number of objects that can be returned in an invocation of the call to the number provided. Setting the parameter to -1 indicates return all contained objects.

**IDLType Interface**

The base interface IDLType (shown below) is inherited by all Interface Repository objects that represent OMG IDL types. It provides access to the TypeCode describing the type, and is used in defining other interfaces wherever definitions of IDL types must be referenced.

```idl
module CORBA {
    interface IDLType : IRObjec {          
        readonly attribute TypeCode type;
    };
};
```

The `type` attribute describes the type defined by an object derived from IDLType.
Repository Interface

Repository (shown below) is an interface that provides global access to the Interface Repository. The Repository object can contain constants, typedefs, exceptions, interfaces, value types, value boxes, native types, and modules. As it inherits from Container, it can be used to look up any definition (whether globally defined or defined within a module or an interface) either by name or by id.

Since the Repository derives only from Container and not from Contained, it does not have a RepositoryId associated with it. By default, it is deemed to have the RepositoryId “” (the empty string) for purposes of assigning a value to the defined_in field of the description structure of ModuleDef, InterfaceDef, ValueDef, ValueBoxDef, TypedefDef, ExceptionDef, and ConstantDef that are contained immediately in the Repository object.

```corba
module CORBA {
    interface Repository : Container {
        Contained lookup_id (in RepositoryId search_id);
        TypeCode get_canonical_typecode(in TypeCode tc);
        PrimitiveDef get_primitive (in PrimitiveKind kind);
    }
};
```

The `lookup_id` operation is used to look up an object in a Repository, given its RepositoryId. If the Repository does not contain a definition for `search_id`, a nil object reference is returned.

The `get_canonical_typecode` operation looks up the TypeCode in the Interface Repository and returns an equivalent TypeCode that includes all repository IDs, names, and member_names. If the top level TypeCode does not contain a RepositoryId, such as array and sequence TypeCodes, or TypeCodes from older ORBs, or if it contains a RepositoryId that is not found in the target Repository, then a new TypeCode is constructed by recursively calling `get_canonical_typecode` on each member TypeCode of the original TypeCode.

The `get_primitive` operation returns a reference to a PrimitiveDef with the specified kind attribute. All PrimitiveDefs are immutable and are owned by the Repository.

ModuleDef Interface

A ModuleDef (shown below) can contain constants, typedefs, exceptions, interfaces, value types, value boxes, native types, and other module objects.
module CORBA {
    interface ModuleDef : Container, Contained {
    }
}

struct ModuleDescription {
    Identifier    name;
    RepositoryId   id;
    RepositoryId   defined_in;
    VersionSpec    version;
    }
}

The inherited `describe` operation for a ModuleDef object returns a ModuleDescription.

**ConstantDef Interface**

A ConstantDef object (shown below) defines a named constant.

module CORBA {
    interface ConstantDef : Contained {
        readonly attribute TypeCode      type;
        readonly attribute IDLType       type_def;
        readonly attribute any           value;
    }
}

struct ConstantDescription {
    Identifier    name;
    RepositoryId   id;
    RepositoryId   defined_in;
    VersionSpec    version;
    TypeCode       type;
    any            value;
    }
}

`type` Specifies the TypeCode describing the type of the constant. The type of a constant must be one of the simple types (long, short, float, char, string, octet, and so on).

`type_def` Identifies the definition of the type of the constant.
value

Contains the value of the constant, not the computation of the value (for example, the fact that it was defined as “1+2”).

The `describe` operation for a `ConstantDef` object returns a `ConstantDescription`.

**TypedefDef Interface**

A `TypedefDef` (shown below) is an abstract interface used as a base interface for all named nonobject types (structures, unions, enumerations, and aliases). The `TypedefDef` interface is not inherited by the definition objects for primitive or anonymous types.

```cpp
module CORBA {
    interface TypedefDef : Contained, IDLType {
    }
    struct TypeDescription {
        Identifier              name;
        RepositoryId            id;
        RepositoryId            defined_in;
        VersionSpec             version;
        TypeCode                type;
    }
};
```

The inherited `describe` operation for interfaces derived from `TypedefDef` returns a `TypeDescription`.

**StructDef**

A `StructDef` (shown below) represents an OMG IDL structure definition. It contains the members of the struct.

```cpp
module CORBA {
    struct StructMember {
        Identifier       name;
        TypeCode          type;
        IDLType           type_def;
    }
    typedef sequence <StructMember> StructMemberSeq;
};
```
interface StructDef : TypedefDef, Container{
  readonly attribute StructMemberSeq members;
};

The `members` attribute contains a description of each structure member.

The inherited `type` attribute is a `tk_struct` TypeCode describing the structure.

**UnionDef**

A UnionDef (shown below) represents an OMG IDL union definition. It contains the members of the union.

module CORBA {
  struct UnionMember {
    Identifier name;
    any label;
    TypeCode type;
    IDLType type_def;
  };
  typedef sequence <UnionMember> UnionMemberSeq;
}

interface UnionDef : TypedefDef, Container {
  readonly attribute TypeCode discriminator_type;
  readonly attribute IDLType discriminator_type_def;
  readonly attribute UnionMemberSeq members;
};

`discriminator_type` and `discriminator_type_def`

Describes and identifies the union’s discriminator type.

The `members` attribute contains a description of each union member. The label of each UnionMemberDescription is a distinct value of the discriminator_type. Adjacent members can have the same name. Members with the same name must also have the same type. A label with type octet and value 0 (zero) indicates the default union member.

The inherited `type` attribute is a `tk_union` TypeCode describing the union.
EnumDef

An EnumDef (shown below) represents an OMG IDL enumeration definition.

```corba
module CORBA {
    typedef sequence <Identifier> EnumMemberSeq;
    interface EnumDef : TypedefDef {
        readonly attribute EnumMemberSeq members;
    };
};
```

`members`
Contains a distinct name for each possible value of the enumeration.
The inherited `type` attribute is a `tk_enum` TypeCode describing the enumeration.

AliasDef

An AliasDef (shown below) represents an OMG IDL typedef that aliases another definition.

```corba
module CORBA {
    interface AliasDef : TypedefDef {
        readonly attribute IDLType original_type_def;
    };
};
```

`original_type_def`
Identifies the type being aliased.
The inherited `type` attribute is a `tk_alias` TypeCode describing the alias.

PrimitiveDef

A PrimitiveDef (shown below) represents one of the OMG IDL primitive types. Because
primitive types are unnamed, this interface is not derived from TypedefDef or Contained.

```corba
module CORBA {
    enum PrimitiveKind {
        pk_null, pk_void, pk_short, pk_long, pk_ushort, pk_ulong,
        pk_float, pk_double, pk_boolean, pk_char, pk_octet,
        pk_any, pk_TypeCode, pk_Principal, pk_string, pk_objref,
        pk_longlong, pk_ulonglong, pk_longdouble, pk_wchar, pk_wstring,
    };
};
```
kind

Indicates which primitive type the PrimitiveDef represents. There are no PrimitiveDefs with kind `pk_null`. A PrimitiveDef with kind `pk_string` represents an unbounded string. A PrimitiveDef with kind `pk_objref` represents the OMG IDL type `Object`. A PrimitiveDef with kind `pk_value_base` represents the IDL type `ValueBase`.

The inherited `type` attribute describes the primitive type.

All PrimitiveDefs are owned by the Repository. References to them are obtained using `Repository::get_primitive`.

**StringDef**

A StringDef represents an IDL bounded string type. The unbounded string type is represented as a PrimitiveDef. As string types are anonymous, this interface is not derived from TypedefDef or Contained.

```idl
module CORBA {
    interface StringDef : IDLType {
        attribute unsigned long bound;
    };
};
```

The `bound` attribute specifies the maximum number of characters in the string and must not be zero.

The inherited `type` attribute is a `tk_string` TypeCode describing the string.

**WstringDef**

A WstringDef represents an IDL wide string. The unbounded wide string type is represented as a PrimitiveDef. As wide string types are anonymous, this interface is not derived from TypedefDef or Contained.
module CORBA {
    interface WstringDef : IDLType {
        attribute unsigned long bound;
    };
};

The `bound` attribute specifies the maximum number of wide characters in a wide string, and must not be zero.

The inherited `type` attribute is a `tk_wstring` TypeCode describing the wide string.

**ExceptionDef**

An ExceptionDef (shown below) represents an exception definition. It can contain structs, unions, and enums.

module CORBA {
    interface ExceptionDef : Contained, Container {
        readonly attribute TypeCode              type;
        readonly attribute StructMemberSeq       members;
    };

    struct ExceptionDescription {
        Identifier name;
        RepositoryId id;
        RepositoryId defined_in;
        VersionSpec version;
        TypeCode type;
    };
};

**tk_except** TypeCode that describes the exception.

members

Describes any exception members.

The `describe` operation for a ExceptionDef object returns an ExceptionDescription.
AttributeDef

An AttributeDef (shown below) represents the information that defines an attribute of an interface.

```cpp
module CORBA {
    enum AttributeMode {ATTR_NORMAL, ATTR_READONLY};

    interface AttributeDef : Contained {
        readonly       attribute TypeCode          type;
        attribute IDLType           type_def;
        attribute AttributeMode     mode;
    };

    struct AttributeDescription {
        Identifier         name;
        RepositoryId       id;
        RepositoryId       defined_in;
        VersionSpec        version;
        TypeCode           type;
        AttributeMode      mode;
    };
};
```

type
Provides the TypeCode describing the type of this attribute.

type_def
Identifies the object that defines the type of this attribute.

mode
Specifies read only or read/write access for this attribute.

The `describe` operation for an AttributeDef object returns an AttributeDescription.

OperationDef

An OperationDef (shown below) represents the information needed to define an operation of an interface.

```cpp
module CORBA {
    enum OperationMode {OP_NORMAL, OP_ONEWAY};
```
enum ParameterMode {PARAM_IN, PARAM_OUT, PARAM_INOUT};

struct ParameterDescription {
    Identifier name;
    TypeCode type;
    IDLType type_def;
    ParameterMode mode;
};
typedef sequence <ParameterDescription> ParDescriptionSeq;

typedef Identifier ContextIdentifier;
typedef sequence <ContextIdentifier> ContextIdSeq;

typedef sequence <ExceptionDef> ExceptionDefSeq;
typedef sequence <ExceptionDescription> ExcDescriptionSeq;

interface OperationDef : Contained {
    readonly attribute TypeCode result;
    readonly attribute IDLType result_def;
    readonly attribute ParDescriptionSeq params;
    readonly attribute OperationMode mode;
    readonly attribute ContextIdSeq contexts;
    readonly attribute ExceptionDefSeq exceptions;
};

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

A TypeCode that describes the type of the value returned by the operation.

result_def

Identifies the definition of the returned type.

params

Describes the parameters of the operation. It is a sequence of ParameterDescription structures. The order of the ParameterDescriptions in the sequence is significant. The name member of each structure provides the parameter name. The type member is a TypeCode describing the type of the parameter. The type_def member identifies the definition of the type of the parameter. The mode member indicates whether the parameter is an in, out, or inout parameter.

mode

The operation’s mode is either oneway (that is, no output is returned) or normal.

contexts

Specifies the list of context identifiers that apply to the operation.

exceptions

Specifies the list of exception types that can be raised by the operation.

The inherited describe operation for an OperationDef object returns an OperationDescription. The inherited describe_contents operation provides a complete description of this operation, including a description of each parameter defined for this operation.

InterfaceDef

An InterfaceDef object (shown below) represents an interface definition. It can contain constants, typedefs, exceptions, operations, and attributes.

module CORBA { interface InterfaceDef; 
      typedef sequence <InterfaceDef> InterfaceDefSeq; 
      typedef sequence <RepositoryId> RepositoryIdSeq; 
      typedef sequence <OperationDescription> OpDescriptionSeq; 
      typedef sequence <AttributeDescription> AttrDescriptionSeq; 

      interface InterfaceDef : Container, Contained, IDLType { 

      }
readonly attribute InterfaceDefSeq base_interfaces;
readonly attribute boolean is_abstract;

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;
    boolean is_abstract;
};

FullInterfaceDescription describe_interface();

struct InterfaceDescription {
    Identifier name;
    RepositoryId id;
    RepositoryId defined_in;
    VersionSpec version;
    RepositoryIdSeq base_interfaces;
    boolean is_abstract;
};

The base_interfaces attribute lists all the interfaces from which this interface inherits.
The is_abstract attribute is TRUE if the interface is an abstract interface type.
The is_a operation returns TRUE if the interface on which it is invoked either is identical to or inherits, directly or indirectly, from the interface identified by its interface_id parameter. Otherwise, it returns FALSE.
The describe_interface operation returns a FullInterfaceDescription describing the interface, including its operations and attributes. The operations and attributes fields of the
FullInterfaceDescription structure include descriptions of all of the operations and attributes in the transitive closure of the inheritance graph of the interface being described.

The inherited describe operation for an InterfaceDef returns an InterfaceDescription.

The inherited contents operation returns the list of constants, typedefs, and exceptions defined in this InterfaceDef and the list of attributes and operations either defined or inherited in this InterfaceDef. If the exclude_inherited parameter is set to TRUE, only attributes and operations defined within this interface are returned. If the exclude_inherited parameter is set to FALSE, all attributes and operations are returned.
Joint Client/Servers

This chapter describes programming requirements for CORBA joint client/servers and the C++ BEAWrapper Callbacks API.

**Note:** The BEA Tuxedo CORBA Java client and BEA Tuxedo CORBA Java client ORB were deprecated in Tuxedo 8.1 and are no longer supported in Tuxedo 9.x. All BEA Tuxedo CORBA Java client and BEA Tuxedo CORBA Java client ORB text references, associated code samples, etc. should only be used:

- to help implement/run third party Java ORB libraries, and
- for programmer reference only.

Technical support for third party CORBA Java ORBs should be provided by their respective vendors. BEA Tuxedo does not provide any technical support or documentation for third party CORBA Java ORBs.

This topic includes the following sections:

- **Introduction.** This section describes:
  - Main Program and Server Initialization
  - Servants
  - Servant Inheritance from Skeletons
  - Callback Object Models Supported
  - Configuring Servers to Call Remote Joint Client/Server Objects
  - Preparing Callback Objects Using CORBA (C++ Joint Client/Servers Only)
Preparing Callback Objects Using BEAWrapper Callbacks

C++ BEAWrapper Callbacks Interface API

Introduction

For either a BEA Tuxedo CORBA client or joint client/server (that is, a client that can receive and process object invocations), the programmer writes the client main(). The main() uses BEA Tuxedo CORBA environmental objects to establish connections, set up security, and start transactions.

BEA Tuxedo clients invoke operations on objects. In the case of DII, client code creates the DII Request object and then invokes one of two operations on the DII Request. In the case of static invocation, client code performs the invocation by performing what looks like an ordinary invocation (which ends up calling code in the generated client stub). Additionally, the client programmer uses ORB interfaces defined by OMG, and BEA Tuxedo CORBA environmental objects that are supplied with the BEA Tuxedo software, to perform functions unique to BEA Tuxedo.

For BEA Tuxedo joint client/server applications, the client code must be structured so that it can act as a server for callback BEA Tuxedo objects. Such clients do not use the TP Framework and are not subject to BEA Tuxedo system administration. Besides the programming implications, this means that CORBA joint client/servers do not have the same scalability and reliability as BEA Tuxedo CORBA servers, nor do they have the state management and transaction behavior available in the TP Framework. If a user wants to have those characteristics, the application must be structured in such a way that the object implementations are in a BEA Tuxedo CORBA server, rather than in a client.

The following sections describe the mechanisms you use to add callback support to a BEA Tuxedo client. In some cases, the mechanisms are contrasted with the BEA Tuxedo server mechanisms that use the TP Framework.

Main Program and Server Initialization

In a BEA Tuxedo server, you use the buildobjserver command to create the main program for the C++ server. (Java servers are not supported in release 8.0 and later of BEA Tuxedo.) Server main program takes care of all BEA Tuxedo- and CORBA-related initialization of the server functions. However, since you implement the Server object, you have an opportunity to customize the way in which the server application is initialized and shut down. The server main program automatically invokes methods on the Server object at the appropriate times.
In contrast, for a BEA Tuxedo CORBA joint client/server (as for a BEA Tuxedo CORBA client), you create the main program and are responsible for all initialization. You do not need to provide a Server object because you have complete control over the main program and you can provide initialization and shutdown code in any way that is convenient.

The specific initialization needed for a joint client/server is discussed in the section “Servants” on page 11-3.

Servants

Servants (method code) for joint client/servers are very similar to servants for servers. All business logic is written the same way. The differences result from not using the TP Framework. Therefore, the main difference is that you use CORBA functions directly instead of indirectly through the TP Framework.

The Server interface is used in BEA Tuxedo CORBA servers to allow the TP Framework to ask the user to create a servant for an object when the ORB receives a request for that object. However, in joint client/servers, the user program is responsible for creating a servant before any requests arrive; thus, the Server interface is not needed. Typically, the program creates a servant and then activates the object (using the servant and an ObjectId; the ObjectId is possibly system generated) before handing a reference to the object. Such an object might be used to handle callbacks. Thus, the servant already exists and the object is activated before a request for the object arrives.

For C++ joint client/servers, instead of invoking the TP interface to perform certain operations, client servants directly invoke the ORB and POA (which is what the TP interface does internally). Alternately, since much of the interaction with the ORB and POA is the same for all applications, for ease of use, the client library provides a convenience wrapper object that does the same things, using a single operation. For a discussion of how to use the convenience wrapper object, see Callback Object Models Supported and Preparing Callback Objects Using BEAWrapper Callbacks.

Servant Inheritance from Skeletons

In a client that supports callbacks, as well as in a server, you write a implementation class that inherits from the same skeleton class name generated by the IDL compiler (the idl command).

C++ Example of Inheritance from Skeletons

The following is a C++ example, given the IDL:
interface Hospital{ ... };

The skeleton generated by the idl command contains a “skeleton” class, POA_Hospital, that the user-written class inherits from, as in:

class Hospital_i : public POA_Hospital { ... };

In a server, the skeleton class inherits from the TP Framework class Tobj_ServantBase, which in turn inherits from the predefined PortableServer::ServantBase.

The inheritance tree for a callback object implementation in a joint client/server is different than that in a server. The skeleton class does not inherit from the TP Framework class Tobj_ServantBase, but instead inherits directly from PortableServer::ServantBase. This behavior is achieved by specifying the -P option in the idl command.

Not having the Tobj_ServantBase class in the inheritance tree for a servant means that the servant does not have activate_object and deactivate_object methods. In a server, these methods are called by the TP Framework to dynamically initialize and save a servant’s state before invoking a method on the servant. For a client that supports callbacks, you must write code that explicitly creates a servant and initializes a servant’s state.

**Callback Object Models Supported**

BEA Tuxedo CORBA supports four kinds of callback objects and provides wrappers for the three that are most common. These objects correspond to three combinations of POA policies. The POA policies control both the types of objects and the types of object references that are possible. The POA policies that are applicable are:

- The LifeSpanPolicy, which controls how long an object reference is valid.
- The IdAssignmentPolicy, which controls who assigns the ObjectId—the user or the system.

These objects are explained primarily in terms of their behavioral characteristics rather than in details about how the ORB and the POA handle them. Those details are discussed in the next sections, using either direct ORB and POA calls (which requires a little extra knowledge of CORBA servers) or using the BEAWrapper Callbacks interface, which hides the ORB and POA calls (for users who do not care about the details).

- Transient/SystemId—object references are valid only for the life of the client process. The ObjectId is not assigned by the client application, but is a unique value assigned by the system. This type of object is useful for invocations that a client wants to receive only until
the client terminates. (The corresponding POA LifeSpanPolicy value is `TRANSIENT` and the IdAssignmentPolicy is `SYSTEM_ID`.)

- **Persistent/SystemId**—object references are valid across multiple activations. The `ObjectId` is not assigned by the client application, but is a unique value assigned by the system. This type of object and object reference is useful when the client goes up and down over a period of time. When the client is up, it can receive callback objects on that particular object reference.

  Typically, the client will create the object reference once, save it in its own permanent storage area, and reactivate the servant for that object every time it comes up. If used with a BEA Tuxedo CORBA Notification Service application, for example, these are callbacks that correspond to the concept of a persistent subscription; that is, the Notification Service remembers the callback reference and delivers events any time the client is up and declares that it is again ready to receive events. This allows notification service subscriptions to survive client failures or offline-time. (The corresponding POA policy values are `PERSISTENT` and `SYSTEM_ID`.)

- **Persistent/UserId**—this is the same as Persistent/SystemId with the exception that the `ObjectId` has to be assigned by the client application. Such an `ObjectId` might be, for example, a database key meaningful only to the client. (The corresponding POA policy values are `PERSISTENT` and `USER_ID`.)

**Notes:** The Transient/UserId policy combination is not considered particularly important. It is possible for users to provide for themselves by using the POA in a manner analogous to either of the persistent cases, but the BEA Tuxedo wrappers do not provide special help to do so.

For BEA Tuxedo CORBA native joint client/servers, neither of the Persistent policies is supported, only the Transient policy.

### Configuring Servers to Call Remote Joint Client/Server Objects

In order for a BEA Tuxedo server to call remote joint client/server objects, that is, joint client/server objects located outside the BEA Tuxedo domain, the server must be configured to enable outbound IIOP. This capability is enabled by specifying the `-O` (uppercase letter O) option in the IIOP Server Listener (ISL) server command. Setting the `-O` option enables outbound invokes (outbound IIOP) on joint client/server objects that are not connected to an IIOP Listener Handler (ISH).
You set ISL command options in the `SERVERS` section of the server’s `UBBCONFIG` file. Because support for outbound IIOP requires a small amount of extra resources, the default is outbound IIOP disabled. For more information, see “Using the ISL Command to Configure Outbound IIOP” in *Setting Up a BEA Tuxedo Application* and “ISL(1)” in the *BEA Tuxedo Command Reference*.

**Preparing Callback Objects Using CORBA (C++ Joint Client/Servers Only)**

To set up BEA Tuxedo C++ callback objects using CORBA, the client must do the following:

1. Establish a connection with a POA with the appropriate policies for the callback object model. (This can be the root POA, available by default, or it may require creating a new POA.)

2. Create a servant (that is, an instance of the C++ implementation class for the interface).

3. Inform the POA that the servant is ready to accept requests on the callback object. Technically, this means the client activates the object in the POA (that is, puts the servant and the `ObjectId` into the POA’s Active Object Map).

4. Tell the POA to start accepting requests from the network (that is, activate the POA itself).

5. Create an object reference for the callback object.

6. Give out the object reference. This usually happens by making an invocation on another object with the callback object reference as a parameter (that is, the parameter is a callback object). That other object will then invoke the callback object (perform a callback invocation) at some later time.

Assuming that the client already has obtained a reference to the ORB, performing this task takes four interactions with the ORB and the POA. It might look like the model show in *Listing 11-1*. In this model, only the Root POA is needed.

**Listing 11-1  Transient/SystemId Model**

```c++
// Create a servant for the callback Object
Catcher_i* my_catcher_i = new Catcher_i();

// Get root POA reference and activate the POA
1  CORBA::Object_var oref =
   orb->resolve_initial_references("RootPOA");
```
To use the Persistent/UserId model, there are some additional steps required when creating a POA. Further, the ObjectId is specified by the client, and this requires more steps. It might look like the model shown in Listing 11-2.

Listing 11-2   Persistent/UserId Model

```c++
Catcher_i* my_catcher_i = new Catcher_i();
const char* oid_str = "783";
PortableServer::objectId_var oid = PortableServer::string_to_objectId(oid_str);
// Find root POA
ORB::Object_var oref = orb->resolve_initial_references("RootPOA");
PortableServer::POA_var root_poa = PortableServer::POA::_narrow(oref);
// Create and activate a Persistent/UserId POA
CORBA::PolicyList policies(2);
policies.length(2);
policies[0] = root_poa->create_lifespan_policy(PortableServer::PERSISTENT);
policies[1] = root_poa->create_id_assignment_policy(PortableServer::USER_ID);
PortableServer::POA_var my_poa_ref = root_poa->create_POA("my_poa_ref", root_poa->the_POAManager(), policies);
root_poa->the_POAManager()->activate();
// Create object reference for callback Object
```
All the interfaces and operations described here are standard CORBA interfaces and operations.

Preparation Callback Objects Using BEAWrapper Callbacks

You can use the BEAWrapper callbacks API with to write either C++ joint client/servers.

Using BEAWrapper Callbacks With C++

Because the code required for callback objects is nearly identical for every client that supports callbacks, you may find it convenient to use the BEAWrappers provided in the library provided for joint client/servers.

The BEAWrappers are described in IDL, as shown in Listing 11-3.

Listing 11-3  BEAWrapper IDL

```
// File: BEAWrapper
#ifndef _BEA_WRAPPER__IDL_
#define _BEA_WRAPPER__IDL_
#include <orb.idl>
#include <PortableServer.idl>
#pragma prefix "beasys.com"
module BEAWrapper {
    interface Callbacks {
        exception ServantAlreadyActive{ };  
        exception ObjectAlreadyActive { };  
        exception NotInRequest{ };  
```
// set up transient callback Object
// -- prepare POA, activate object, return objref
Object start_transient(
   in PortableServer::Servant Servant,
   in CORBA::RepositoryId rep_id)
raises (ServantAlreadyActive);

// set up persistent/systemid callback Object
Object start_persistent_systemid(
   in PortableServer::Servant servant,
   in CORBA::Repository rep_id,
   out string stroid)
raises (ServantAlreadyActive);

// reinstate set up for persistent/systemid
// callback object
Object restart_persistent_systemid(
   in PortableServer::Servant servant,
   in CORBA::RepositoryId rep_id,
   in string stroid)
raises (ServantAlreadyActive, ObjectAlreadyActive);

// set up persistent/userid callback Object
Object start_persistent_userid(
   in PortableServer::Servant servant,
   in CORBA::RepositoryId rep_id,
   in string stroid)
raises (ServantAlreadyActive, ObjectAlreadyActive);

// stop servicing a particular callback Object
// with the given servant
void stop_object( in PortableServer::Servant servant);

// Stop all callback Object processing
void stop_all_objects();

// get oid string for the current request
string get_string_oid() raises (NotInRequest);
The BEAwrappers are described in C++ as shown in Listing 11-4.

Listing 11-4  C++ Declarations (in beawrapper.h)

```c++
#ifndef _BEAWRAPPER_H_
#define _BEAWRAPPER_H_
#include <PortableServer.h>
class BEAWrapper{
    class Callbacks{
        public:
            Callbacks (CORBA::ORB_ptr init_orb);
            CORBA::Object_ptr start_transient (Portableserver::Servant servant, const char * rep_id);
            CORBA::Object_ptr start_persistent_systemid (Portableserver::Servant servant, const char * rep_id, char * & stroid);
            CORBA::Object_ptr restart_persistent_systemid (Portableserver::Servant servant, const char * rep_id, const char * stroid);
            CORBA::Object_ptr start_persistent_userid (Portableserver::Servant servant, const char * rep_id, const char * stroid);
            void stop_object(Portableserver::Servant servant);
            char* get_string_oid ();
            void stop_all_objects();
```
C++ BEAWrapper Callbacks Interface API

This C++ BEAWrapper Callbacks interface API is described in the following sections.

Callbacks

Synopsis
Returns a reference to the Callbacks interface.

C++ Binding

BEAWrapper::Callbacks( CORBA::ORB_ptr init_orb);

Argument

init_orb
The ORB to be used for all further operations.

Exception

CORBA::IMP_LIMIT
The BEAWrapper::Callbacks class has already be instantiated with an ORB pointer. Only one instance of this class can be used in a process. Users who need additional flexibility should use the POA directly.
Description

The constructor returns a reference to the Callbacks interface. Only one such object should be
created for the process, even if multiple threads are used. Using more than one such object will
result in undefined behavior.

Return Value

A reference to the Callbacks object.

\**start_transient**

Synopsis

Activates an object, sets the ORB and the POA to the proper state, and returns an object reference
to the activated object.

IDL

\[
\text{Object start_transient( in PortableServer::Servant servant, }
\text{in CORBA::RepositoryId rep_id)}\]
\[
\text{raises ( ServantAlreadyActive );}
\]

C++ Binding

\[
\text{CORBA::Object_ptr start_transient(}
\text{PortableServer::Servant servant,}
\text{const char* rep_id);}\]

Arguments

- **servant**
  
  An instance of the C++ implementation class for the interface.

- **rep_id**
  
  The repository id of the interface.

Exceptions

- **ServantAlreadyActive**
  
  The servant is already being used for a callback. A servant can be used only for a callback
  with a single ObjectId. To receive callbacks on objects containing different ObjectIds,
you must create different servants and activate them separately. The same servant can be
reused only if a stop_object operation tells the system to stop using the servant for its original ObjectId.

CORBA::BAD_PARAM
The repository ID was a NULL string or the servant was a NULL pointer.

Description
This operation performs the following actions:

- Activates an object using the Servant supplied to service objects of the type rep_id, using an ObjectId generated by the system.
- Sets the ORB and the POA into the state in which they will accept requests on that object.
- Returns an object reference to the activated object. The returned object reference will be valid only until the termination of the client or until the callback servant is halted by the user via the stop_object operation; after that, invocations on that object reference are no longer valid and can never be made valid.

Return Value
CORBA::Object_ptr
A reference to the object that was created with the ObjectId generated by the system and the rep_id provided by the user. The object reference will need to be converted to a specific object type by invoking the narrow() operation defined for the specific object. The caller is responsible for releasing the object when the conversion is done.

**start_persistent_systemid**

Synopsis
Activates an object, sets the ORB and the POA to the proper state, sets the output parameter stroid, and returns an object reference to the activated object.

IDL

```idl
Object start_persistent_systemid(
    in PortableServer::Servant servant,
    in CORBA::RepositoryId rep_id,
    out string stroid)
raises ( ServantAlreadyActive );
```
C++ Binding

```cpp
CORBA::Object_ptr start_persistent_systemid(
    PortableServer::Servant          servant,
const char*                      rep_id,
char*&                            stroid);
```

JArguments

- **servant**
  
  An instance of the C++ implementation class for the interface.

- **rep_id**
  
  The repository ID of the interface.

- **stroid**
  
  This argument is set by the system and is opaque to the user. The client will use it when it reactivates the object at a later time (using `restart_persistent_systemid`), most likely after the client process has terminated and restarted.

Exceptions

- **ServantAlreadyActive**
  
  The servant is already being used for a callback. A servant can be used only for a callback with a single `ObjectId`. To receive callbacks on objects containing different `ObjectId`s, you must create different servants and activate them separately. The same servant can be reused only if a `stop` operation tells the system to stop using the servant for its original `ObjectId`.

- **CORBA::BAD_PARAMETER**
  
  The repository ID was a NULL string or the servant was a NULL pointer.

- **CORBA::IMP_LIMIT**
  
  In addition to other system reasons for this exception, a reason unique to this situation is that the joint client/server was not initialized with a port number; therefore, a persistent object reference cannot be created.

Description

This operation performs the following actions:

- Activates an object using the `Servant` supplied to service objects of the type `rep_id`, using an `ObjectId` generated by the system.

- Sets the ORB and the POA into the state in which they will accept requests on that object.
• Sets the output parameter stroid to the stringified version of an ObjectId assigned by the system.

• Returns an object reference to the activated object. The returned object reference will be valid even after termination of the client. That is, if the client terminates, restarts again, and then activates a servant with the same rep_id and for the same ObjectId, the servant will accept requests made on that same object reference. Since the ObjectId was generated by the system, the application has to save that ObjectId.

Return Value
CORBA::Object_ptr
An object reference created with the ObjectId generated by the system and the rep_id provided by the user. The object reference will need to be converted to a specific object type by invoking the _narrow() operation defined for the specific object. The caller is responsible for releasing the object when the conversion is done.

restart_persistent_systemid

Synopsis
Activates an object, sets the ORB and the POA to the proper state, and returns an object reference to the activated object.

IDL
Object restart_persistent_systemid(
   in PortableServer::Servant   servant,
   in CORBA::RepositoryId       rep_id,
   in string                    stroid)
raises (ServantAlreadyActive, ObjectAlreadyActive);

C++ Binding
CORBA::Object_ptr restart_persistent_systemid(
   PortableServer::Servant   servant,
   const char*               rep_id
   const char*               stroid);
Arguments

servant
An instance of the C++ implementation class for the interface.

rep_id
The repository ID of the interface.

stroid
The stringified version of the ObjectId provided by the user to be set in the object reference being created. It must have been returned from a previous call on start_persistent_systemid.

Exceptions

ServantAlreadyActive
The servant is already being used for a callback. A servant can be used only for a callback with a single ObjectId. To receive callbacks on objects containing different ObjectIds, you must create different servants and activate them separately. The same servant can be reused only if a stop_object operation tells the system to stop using the servant for its original ObjectId.

ObjectAlreadyActive
The stringified ObjectId is already being used for a callback. A given ObjectId can have only one servant associated with it. If you wish to change to a different servant, you must first invoke stop_object with the servant currently in use.

CORBA::BAD_PARAM
The repository ID was a NULL string or the servant was a NULL pointer or the ObjectId supplied was not previously assigned by the system.

CORBA::IMP_LIMIT
In addition to other system reasons for this exception, a reason unique to this situation is that the joint client/server was not initialized with a port number; therefore, a persistent object reference cannot be created.

Description

This operation performs the following actions:

- Activates an object using the Servant supplied to service objects of the type rep_id, using the supplied stroid (a stringified ObjectId), which must have been obtained by a previous call on start_persistent_systemid.
- Sets the ORB and the POA into the state in which they will accept requests on that object.
- Returns an object reference to the object activated.
The reactivation would be done using the `restart_persistent_systemid` method.

**Return Value**

CORBA::Object_ptr

An object reference created with the stringified `ObjectId stroid` and the `rep_id` provided by the user. The object reference will need to be converted to a specific object type by invoking the `_narrow()` operation defined for the specific object. The caller is responsible for releasing the object when done.

**start_persistent_userid**

**Synopsis**

Activates an object, sets the ORB and the POA to the proper state, and returns an object reference to the activated object.

**IDL**

```
Object start_persistent_userid(
    portableServer::Servant a_servant,
    in CORBA::RepositoryId rep_id,
    in string stroid)
raises ( ServantAlreadyActive, ObjectAlreadyActive );
```

**C++ Binding**

```cpp
CORBA::Object_ptr start_persistent_userid (
    PortableServer::Servant servant,
    const char* rep_id,
    const char* stroid);
```

**Arguments**

- **servant**
  
  An instance of the C++ implementation class for the interface.

- **rep_id**
  
  The repository ID of the interface.
stroid
The stringified version of an ObjectId provided by the user to be set in the object reference being created. The stroid holds application-specific data and is opaque to the ORB.

Exceptions

ServantAlreadyActive
The servant is already being used for a callback. A servant can be used only for a callback with a single ObjectId. To receive callbacks on objects containing different ObjectIds, you must create different servants and activate them separately. The same servant can be reused only if a stop_object operation tells the system to stop using the servant for its original ObjectId.

ObjectAlreadyActive
The stringified ObjectId is already being used for a callback. A given ObjectId can have only one servant associated with it. If you wish to change to a different servant, you must first invoke stop_object with the servant currently in use.

CORBA::BAD_PARAM
The repository ID was a NULL string or the servant was a NULL pointer.

CORBA::IMP_LIMIT
In addition to other system reasons for this exception, a reason unique to this situation is that the joint client/server was not initialized with a port number; therefore, a persistent object reference cannot be created.

Description
This operation performs the following actions:

- Activates an object using the Servant supplied to service objects of the type rep_id, using the object Id stroid.
- Sets the ORB and the POA into the state in which they will accept requests on that object.
- Returns an object reference to the activated object. The returned object reference will be valid even after termination of the client. That is, if the client terminates, and restarts again, and then activates a servant with the same rep_id and for the same ObjectId, the servant will accept requests made on that same object reference.

Return Value

CORBA::Object_ptr
An object reference created with the stringified ObjectId stroid and the rep_id provided by the user. The object reference will need to be converted to a specific object
type by invoking the \texttt{\_narrow()} operation defined for the specific object. The caller is responsible for releasing the object when the conversion is done.

\textbf{stop\_object}

\textbf{Synopsis}

Tells the ORB to stop accepting requests on the object that is using the given servant.

\textbf{IDL}

\begin{verbatim}
void stop_object( in PortableServer::Servant servant);
\end{verbatim}

\textbf{C++ Binding}

\begin{verbatim}
void stop_object(PortableServer::Servant servant);
\end{verbatim}

\textbf{Argument}

\begin{verbatim}
servant
\end{verbatim}

\begin{itemize}
\item An instance of the C++ implementation class for the interface. The association between this servant and its ObjectId will be removed from the Active Object Map.
\end{itemize}

\textbf{Exceptions}

None.

\textbf{Description}

This operation tells the ORB to stop accepting requests on the given servant. It does not matter what state the servant is in, activated or deactivated; no error is reported.

\textbf{Note:}

If you do an invocation on a callback object after you call the \texttt{stop\_object} operation, the \texttt{OBJECT\_NOT\_EXIST} exception is returned to the caller. This is because the \texttt{stop\_object} operation, in effect, deletes the object.

\textbf{Return Value}

None.

\textbf{stop\_all\_objects}

\textbf{Synopsis}

Tells the ORB to stop accepting requests on all servants.
**stop_all_objects**

**IDL**

```idl
void stop_all_objects ();
```

**C++ Binding**

```cpp
void stop_all_objects ();
```

**Exceptions**

None.

**Description**

This operation tells the ORB to stop accepting requests on all servants that have been set up in this process.

**Usage Note**

If a client calls the `ORB::shutdown` method, then it must not subsequently call `stop_all_objects`.

**Return Value**

None.

---

**get_string_oid**

**Synopsis**

Requests the string version of the `ObjectId` of the current request.

**IDL**

```idl
string get_string_oid() raises (NotInRequest);
```

**C++ Binding**

```cpp
char* get_string_oid();
```

**Exceptions**

`NotInRequest`

The function was called when the ORB was not in the context of a request (that is, not while the ORB was servicing a request in method code). Do not call this function from client code. It is legal only during the execution of a method of the callback object (that is, the servant).
Description

This operation returns the string version of the ObjectId of the current request.

Return Value

char*

The string version of the ObjectId of the current request. This is the string that was supplied when the object reference was created. The string is meaningful to users only in the case when the object reference was created by the start_persistent_userid function. (That is, the ObjectId created by start_transient and start_persistent_systemid were created by the ORB and has no relationship to the user application.)

~Callbacks

Synopsis

Destroys the callback object.

C++ Binding

BEAWrapper::~Callbacks();

JArguments

None.

Exceptions

None.

Description

This destructor destroys the callback object.

Usage Note

If a client wants to get rid of the wrapper, but not shut down the ORB, then the client must call the stop_all_objects method.

Return Value

None.
Development Commands

For a detailed discussion of BEA Tuxedo development commands, see the *BEA Tuxedo Command Reference*. This document describes all BEA Tuxedo commands and processes.

A PDF file of the *BEA Tuxedo Command Reference* is also provided in the online documentation.
Mapping of OMG IDL Statements to C++

This chapter discusses the mappings from OMG IDL statements to C++.

Note: Some of the information in this chapter is taken from the *Common Object Request Broker: C++ Language Mapping Specification*, June 1999, published by the Object Management Group (OMG). Used with permission of the OMG.

Mappings

OMG IDL-to-C++ mappings are described for the following:

- Data Types
- Strings
- wchars
- wstrings
- Constants
- Enums
- Structs
- Unions
- Sequences
- Arrays
In addition, the following topics are discussed:

- Fixed-length Versus Variable-length User-defined Types
- Using var Classes
- Using out Classes
- Argument Passing Considerations

**Data Types**

Each OMG IDL data type is mapped to a C++ data type or class.
Basic Data Types

The basic data types in OMG IDL statements are mapped to C++ typedefs in the CORBA module, as shown in Table 13-1.

Table 13-1  Basic OMG IDL and C++ Data Types

<table>
<thead>
<tr>
<th>OMG IDL</th>
<th>C++</th>
<th>C++ Out Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>short</td>
<td>CORBA::Short</td>
<td>CORBA::Short_out</td>
</tr>
<tr>
<td>long</td>
<td>CORBA::Long</td>
<td>CORBA::Long_out</td>
</tr>
<tr>
<td>unsigned short</td>
<td>CORBA::UShort</td>
<td>CORBA::UShort_out</td>
</tr>
<tr>
<td>unsigned long</td>
<td>CORBA::ULong</td>
<td>CORBA::ULong_out</td>
</tr>
<tr>
<td>float</td>
<td>CORBA::Float</td>
<td>CORBA::Float_out</td>
</tr>
<tr>
<td>double</td>
<td>CORBA::Double</td>
<td>CORBA::Double_out</td>
</tr>
<tr>
<td>char</td>
<td>CORBA::Char</td>
<td>CORBA::Char_out</td>
</tr>
<tr>
<td>boolean</td>
<td>CORBA::Boolean</td>
<td>CORBA::Boolean_out</td>
</tr>
<tr>
<td>octet</td>
<td>CORBA::Octet</td>
<td>CORBA::Octet_out</td>
</tr>
<tr>
<td>wchar</td>
<td>CORBA::WChar</td>
<td>CORBA::WChart_out</td>
</tr>
</tbody>
</table>

Note: On a 64-bit machine where a long integer is 64 bits, the definition of CORBA::Long would still refer to a 32-bit integer.

Complex Data Types

Object, pseudo-object, and user-defined types are mapped as shown in Table 13-2.
The mapping for strings and UDTs is described in more detail in the following sections.

### Strings

A string in OMG IDL is mapped to `char *` in C++. Both bounded and unbounded strings are mapped to `char *`. CORBA strings in C++ are NULL-terminated and can be used wherever a `char *` type is used.

If a string is contained within another user-defined type, such as a `struct`, it is mapped to a `CORBA::String_var` type. This ensures that each member in the struct manages its own memory.

Strings must be allocated and deallocated using the following member functions in the CORBA class:

- `string_alloc`
- `string_dup`
- `string_free`

**Note:** The `string_alloc` function allocates `len+1` characters so that the resulting string has enough space to hold a trailing NULL character.
**wchars**

OMG IDL defines a wchar data type that encodes wide characters from any character set. As with character data, an implementation is free to use any code set internally for encoding wide characters, though, again, conversion to another form may be required for transmission. The size of wchar is implementation-dependent.

The syntax for defining a wchar is:

```
<wide_char_type> ::= "wchar"
```

A code example for wchar is:

```c
wchar_t wmixed[256];
```

**Note:** The wchar and wstring data types enable users to interact with computers in their native written language. Some languages, such as Japanese and Chinese, have thousands of unique characters. These character sets do not fit within a byte. A number of schemes have been used to support multi-byte character sets, but they have proved to be unwieldy to use. Wide characters and wide strings make it easier to interact with this kind of complexity.

**wstrings**

The wstring data type represents a sequence of wchar, except the wide character NULL. The type wstring is similar to that of type string, except that its element type is wchar instead of char. The actual length of a wstring is set at run time and, if the bounded form is used, must be less than or equal to the bound.

The syntax for defining a wstring is:

```
<wide_string_type> ::= "wstring" "<" <positive_int_const> ">" | "wstring
```

A code example for wstring is:

```c
CORBA::WString_var v_upper = CORBA::wstring_dup(wmixed);
```

wstring types are built in types just like unsigned long, char, string, double, etc. They can be used directly as parameters, typedef'd, used to construct structs, sequences, unions, arrays, and so forth.

**Note:** The wchar and wstring data types enable users to interact with computers in their native written language. Some languages, such as Japanese and Chinese, have thousands of unique characters. These character sets do not fit within a byte. A number of schemes have been used to support multi-byte character sets, but they have proved to be unwieldy...
to use. Wide characters and wide strings make it easier to interact with this kind of complexity.

**Constants**

A constant in OMG IDL is mapped to a C++ `const` definition. For example, consider the following OMG IDL definition:

```idl
// OMG IDL
const string CompanyName = "BEA Systems Incorporated";
module INVENT
{
    const string Name = "Inventory Modules";
    interface Order
    {
        const long MAX_ORDER_NUM = 10000;
    };
}
```

This definition maps to C++ as follows:

```cpp
// C++
const char *const CompanyName = "BEA Systems Incorporated";

class INVENT
{
    static const char *const Name;

    class Order : public virtual CORBA::Object
    {
        static const CORBA::Long MAX_ORDER_NUM;
    };
};
```

Top-level constants are initialized in the generated `.h` include file, but module and interface constants are initialized in the generated client stub modules.
The following is an example of a valid reference to the `MAX_ORDER_NUM` constant, as defined in the previous example:

```cpp
CORBA::Long accnt_id = INVENT::Order::MAX_ORDER_NUM;
```

## Enums

An enum in OMG IDL is mapped to an enum in C++. For example, consider the following OMG IDL definition:

```idl
// OMG IDL
module INVENT
{
enum Reply {ACCEPT, REFUSE};
}
```

This definition maps to C++ as follows:

```cpp
// C++
class INVENT
{

enum Reply {ACCEPT, REFUSE};
};
```

The following is an example of a valid reference to the enum defined in the previous example. You refer to enum as follows:

```cpp
INVENT::Reply accept_reply;
accept_reply = INVENT::ACCEPT;
```

## Structs

A struct in OMG IDL is mapped to a C++ struct.

The generated code for a struct depends upon whether it is fixed-length or variable-length. For more information about fixed-length versus variable-length types, see the section Fixed-length Versus Variable-length User-defined Types.
Fixed-length Versus Variable-length Structs

A variable-length struct contains an additional assignment operator member function to handle assignments between two variable-length structs.

For example, consider the following OMG IDL definition:

```idl
// OMG IDL
module INVENT
{
  // Fixed-length
  struct Date
  
  long year;
  long month;
  long day;
  
};

  // Variable-length
  struct Address
  
  string aptNum;
  string streetName;
  string city;
  string state;
  string zipCode;
  
};
}
```

This definition maps to C++ as follows:

```cpp
// C++
class INVENT
{
  struct Date
  
  CORBA::Long year;
  CORBA::Long month;
  CORBA::Long day;
  
};
```
struct Address
{
    CORBA::String_var aptNum;
    CORBA::String_var streetName;
    CORBA::String_var city;
    CORBA::String_var state;
    CORBA::String_var zipCode;
    Address &operator=(const Address &_obj);
};

Member Mapping
Members of a struct are mapped to the appropriate C++ data type. For basic data types (long, short, and so on), see Table 13-1. For object references, pseudo-object references, and strings, the member is mapped to the appropriate var class:

- CORBA::String_var
- CORBA::Object_var

All other data types are mapped as shown in Table 13-2.

No constructor for a generated struct exists, so none of the members are initialized. Fixed-length structs can be initialized using aggregate initialization. For example:

INVENT::Date a_date = { 1995, 10, 12 };

Variable-length members map to self-managing types; these types have constructors that initialize the member.

Var
A var class is generated for structs. For more information, see the section Using var Classes.

Out
An out class is generated for structs. For more information, see the section Using out Classes.

Unions
A union in OMG IDL is mapped to a C++ class. The C++ class contains the following:
• Constructors
• Destructors
• Assignment operators
• Modifiers for the union value
• Accessors for the union value
• Modifiers and accessors for the union discriminator

For example, consider the following OMG IDL definition:

```idl
union OrderItem switch (long)
{
    case 1: itemStruct itemInfo;
    case 2: orderStruct orderInfo;
    default: ID idInfo;
};
```

This definition maps to C++ as follows:

```cpp
class OrderItem
{
public:
    OrderItem();
    OrderItem(const OrderItem&);
    ~OrderItem();

    OrderItem &operator=(const OrderItem&);

    void _d (CORBA::Long);
    CORBA::Long _d () const;

    void itemInfo (const itemStruct&);
    const itemStruct & itemInfo () const;
    itemStruct & itemInfo ();

    void orderInfo (const orderStruct&);
```
const orderStruct & orderInfo () const;
orderStruct & orderInfo ();

void idInfo (ID);
ID idInfo () const;

...};

The default union constructor does not set a default discriminator value for the union; therefore, you cannot call any union accessor member function until you have set the value of the union. The discriminator is an attribute that is mapped through the _d member function.

**Union Member Accessor and Modifier Member Function Mapping**

For each member in the union, accessor and modifier member functions are generated.

In the following code, taken from the previous example, two member functions are generated for the ID member function:

```c
void idInfo (ID);
ID idInfo () const;
```

In this example, the first function (the modifier) sets the discriminator to the default value and sets the value of the union to the specified ID value. The second function, the accessor, returns the value of the union.

Depending upon the data type of the union member, additional modifier functions are generated. The member functions generated for each data type are as follows:

- Basic data types—short, long, unsigned short, unsigned long, float, double, char, boolean, and octet

  The following example generates two member functions for a basic data type with member name basictype:

  ```c
  void basictype (TYPE); // modifier
  TYPE basictype () const; // accessor
  ```

  For the mapping from an OMG IDL data type to the C++ data type TYPE, see Table 13-1.

- Object and pseudo-object
For object and Typecode types with member name `objtype`, member functions are generated as follows:

```c++
void objtype (TYPE);    // modifier
TYPE objtype () const; // accessor
```

For the mapping from an OMG IDL data type to the C++ data type `TYPE`, see Table 13-1.

The modifier member function does not assume ownership of the specified object reference argument. Instead, the modifier duplicates the object reference or pseudo-object reference. You are responsible for releasing the reference when it is no longer required.

- **Enum**

  For an enum `TYPE` with member name `enumtype`, member functions are generated as follows:

  ```c++
  void enumtype (TYPE);    // modifier
  TYPE enumtype () const; // accessor
  ```

- **String**

  For strings, one accessor and three modifier functions are generated, as follows:

  ```c++
  void stringInfo (char *);                     // modifier 1
  void stringInfo (const char *);               // modifier 2
  void stringInfo (const CORBA::String_var &);  // modifier 3
  const char * stringInfo () const;            // accessor
  ```

  The first modifier assumes ownership of the `char *` parameter passed to it and the union is responsible for calling the `CORBA::string_free` member function on this string when the union value changes or when the union is destroyed.

  The second and third modifiers make a copy of the specified string passed in the parameter or contained in the string var.

  The accessor function returns a pointer to internal memory of the union; do not attempt to free this memory, and do not access this memory after the union value has been changed or the union has been destroyed.

- **Struct, union, sequence, and any**

  For these data types, modifier and accessor functions are generated with references to the type.
type, as follows:

```c++
void reftype (TYPE &);       // modifier
const TYPE & reftype () const; // accessor
TYPE & reftype ();             // accessor
```

The modifier function does not assume ownership of the input `type` parameter; instead, the function makes a copy of the data `type`.

- **Array**

  For an array, the modifier member function accepts an array pointer while the accessor returns a pointer to an array slice, as follows:

```c++
void arraytype (TYPE);           // modifier
TYPE_slice * arraytype () const; // accessor
```

The modifier function does not assume ownership of the input `type` parameter; instead, the function makes a copy of the array.

**Var**

A var class is generated for a union. For more information, see the section Using var Classes.

**Out**

An out class is generated for a union. For more information, see the section Using out Classes.

**Member Functions**

In addition to the accessor and modifiers, the following member functions are generated for an OMG IDL union of type `TYPE` with switch (long) discriminator:

```c++
TYPE();

This is the default constructor for a union. No default discriminator is set by this function, so you cannot access the union until you set the value of the union.

TYPE( const TYPE & From);

This copy constructor deep copies the specified union. Any data in the union parameter is copied. The `From` argument specifies the union to be copied.

~TYPE();

This destructor frees the data associated with the union.


**TYPE &operator=(const TYPE & From);**

This assignment operator copies the specified union. Any existing value in the current union is freed. The From argument specifies the union to be copied.

**void _d (CORBA::Long Discrim);**

This modifier function sets the value of the union discriminant. The Discrim argument specifies the new discriminant. The data type of the argument is determined by the OMG IDL data type specified in the switch statement of the union. For each OMG IDL data type, see Table 13-1 for the C++ data type.

Only use this function to set the discriminant to a value within the same union member. You cannot use this function to implicitly switch between different union members.

These restrictions are illustrated by the following code:

```cpp
union U switch(long) {
  case 1:
  case 2:
    short s;
  case 3:
    int it;
};

short st;
U u;
  u.s(1296);  // member "s" selected
  st = u.s(); // st == 1296
  u._d(2);   // OK: member "s" still selected
  st = u.s(); // st == 1296
  u._d(3);   // BAD_PARAM: selecting a different member

When the _d() modifier is invoked on a new instance of a union, Tuxedo C++ relaxes the "implicit switching" restriction. In this case, no exception is thrown, and the union is not affected.

  U u2;
  u2._d(1); // no exception, union is unchanged
  st = u2.s(); // error! accessing an uninitialized union
  u2.it(1296); // OK: member "it" now selected

**CORBA::Long _d () const;**

This function returns the current discriminant value. The data type of the return value is determined by the OMG IDL data type specified in the switch statement of the union. For each OMG IDL data type, see Table 13-1 for the C++ data type.
Sequences

A sequence in OMG IDL is mapped to a C++ class. The C++ class contains the following:

- **Constructors**
  Each sequence has the following:
  - A default constructor
  - A constructor that initializes each element
  - A copy constructor

- **Destructors**

- **Modifiers for current length (and for maximum, if the sequence is unbounded)**

- **Accessors for current length**

- **Operator[] functions to access or modify sequence elements**

- **Allocation and deallocation member functions**

You must set the length before accessing any elements.

For example, consider the following OMG IDL definition:

```idl
// OMG IDL
module INVENT
{
  ...
  typedef sequence<LogItem>       LogList;
}
```

This definition maps to C++ as follows:

```cpp
// C++
class LogList
{
  public:
    // Default constructor
    LogList();

    // Maximum constructor
    LogList(CORBA::ULong _max);
```
// TYPE * data constructor
LogList
(
    CORBA::ULong _max,
    CORBA::ULong _length,
    LogItem * _value,
    CORBA::Boolean _relse = CORBA_FALSE
);

// Copy constructor
LogList(const LogList&);

// Destructor
~LogList();
LogList & operator=(const LogList&);
CORBA::ULong maximum() const;
void length(CORBA::ULong);
CORBA::ULong length() const;
LogItem & operator[](CORBA::ULong _index);
const LogItem & operator[](CORBA::ULong _index) const;
static LogItem * allocbuf(CORBA::ULong _nelems);
static void freebuf(LogItem *);
);

Sequence Element Mapping

The operator[] functions are used to access or modify the sequence element. These operators return a reference to the sequence element. The OMG IDL sequence base type is mapped to the appropriate C++ data type.

For basic data types, see Table 13-1. For object references, TypeCode references, and strings, the base type is mapped to a generated _ForSeq_var class. The _ForSeq_var class provides the capability to update a string or an object that is stored within the sequence. This generated class has the same member functions and signatures as the corresponding var class. However, this _ForSeq_var class honors the setting of the release parameter in the sequence constructor. The distinction is that the _ForSeq_var class lets users specify the Release flag, thereby allowing users to control the freeing of memory.
All other data types are mapped as shown in Table 13-2.

**Vars**
A var class is generated for a sequence. For more information, see the section Using var Classes.

**Out**
An out class is generated for a sequence. For more information, see the section Using out Classes.

**Member Functions**
For a given OMG IDL sequence \( \text{SEQ} \) with base type \( \text{TYPE} \), the member functions for the generated sequence class are described as follows:

\[ \text{SEQ}(); \]
This is the default constructor for a sequence. The length is set to 0 (zero). If the sequence is unbounded, the maximum is also set to 0 (zero). If the sequence is bounded, the maximum is specified by the OMG IDL type and cannot be changed.

\[ \text{SEQ} \left( \text{CORBA::ULong Max} \right); \]
This constructor is present only if the sequence is unbounded. This function sets the length of the sequence to 0 (zero) and sets the maximum of the buffer to the specified value. The \( \text{Max} \) argument specifies the maximum length of the sequence.

\[ \text{SEQ} \left( \text{CORBA::ULong Max, CORBA::ULong Length, TYPE * Value, CORBA::Boolean Release} \right); \]
This constructor sets the maximum, length, and elements of the sequence. The \( \text{Release} \) flag determines whether elements are released when the sequence is destroyed.

Explanations of the arguments are as follows:

- \( \text{Max} \)
  The maximum value of the sequence. This argument is not present in bounded sequences.

- \( \text{Length} \)
  The current length of the sequence. For bounded sequences, this value must be less than the maximum specified in the OMG IDL type.

- \( \text{Value} \)
  A pointer to the buffer containing the elements of the sequence.

- \( \text{Release} \)
  Determines whether elements are released. If this flag has a value of \( \text{CORBA_TRUE} \), the sequence assumes ownership of the buffer pointed to by the \( \text{Value} \) argument.
If the Release flag is CORBA_TRUE, this buffer must be allocated using the allocbuf member function, because it will be freed using the freebuf member function when the sequence is destroyed.

SEQ(const S& From);
This copy constructor deep copies the sequence from the specified argument. The From argument specifies the sequence to be copied.

~SEQ();
This destructor frees the sequence and, depending upon the Release flag, may free the sequence elements.

SEQ& operator=(const SEQ& From);
This assignment operator deep copies the sequence from the specified sequence argument. Any existing elements in the current sequence are released if the Release flag in the current sequence is set to CORBA_TRUE. The From argument specifies the sequence to be copied.

CORBA::ULong maximum() const;
This function returns the maximum of the sequence. For a bounded sequence, this is the value set in the OMG IDL type. For an unbounded sequence, this is the current maximum of the sequence.

void length(CORBA::ULong Length);
This function sets the current length of the sequence. The Length argument specifies the new length of the sequence. If the sequence is unbounded and the new length is greater than the current maximum, the buffer is reallocated and the elements are copied to the new buffer. If the new length is greater than the maximum, the maximum is set to the new length.

For a bounded sequence, the length cannot be set to a value greater than the maximum.

CORBA::ULong length() const;
This function returns the current length of the sequence.

TYPE & operator[](CORBA::ULong Index);
const TYPE & operator[](CORBA::ULong Index) const;
These accessor functions return a reference to the sequence element at the specified index. The Index argument specifies the index of the element to return. This index cannot be greater than the current sequence length. The length must have been set either using the TYPE * constructor or the length(CORBA::ULong) modifier. If TYPE is an object reference, TypeCode reference, or string, the return type will be a ForSeq_var class.
static TYPE * allocbuf(CORBA::ULong NumElems);

This static function allocates a buffer to be used with the TYPE * constructor. The NumElems argument specifies the number of elements in the buffer to allocate. If the buffer cannot be allocated, NULL is returned.

If this buffer is not passed to the TYPE * constructor with release set to CORBA_TRUE, it should be freed using the freebuf member function.

static void freebuf(TYPE * Value);

This static function frees a TYPE * sequence buffer allocated by the allocbuf function. The Value argument specifies the TYPE * buffer allocated by the allocbuf function. A 0 (zero) pointer is ignored.

Arrays

An array in OMG IDL is mapped to a C++ array definition. For example, consider the following OMG IDL definition:

```
// OMG IDL
module INVENT
{
  ...
  typedef LogItem LogArray[10];
};
```

This definition maps to C++ as follows:

```
// C++
module INVENT
{
  ...
  typedef LogItem LogArray[10];
typedef LogItem LogArray_slice;
static LogArray_slice * LogArray_alloc(void);
static void LogArray_free(LogArray_slice *data);
};
```
**Array Slice**

A slice of an array is an array with all the dimensions of the original array except the first dimension. The member functions for the array-generated classes use a pointer to a slice to return pointers to an array. A typedef for each slice is generated.

For example, consider the following OMG IDL definition:

// OMG IDL
typedef LogItem         LogMultiArray[5][10];

This definition maps to C++ as follows:

// C++
typedef LogItem          LogMultiArray[5][10];
typedef LogItem           LogMultiArray_slice[10];

If you have a one-dimensional array, an array slice is just a type. For example, if you had a one-dimensional array of `long`, an array slice would result in a `CORBA::Long` data type.

**Array Element Mapping**

The type of the OMG IDL array is mapped to the C++ array element type in the same manner as structs. For more information, see the section Member Mapping.

**Vars**

A var class is generated for an array. For more information, see the section Using var Classes.

**Out**

An out class is generated for an array. For more information, see the section Using out Classes.

**Allocation Member Functions**

For each array, there are two static functions for array allocation and deallocation. For a given OMG IDL type `TYPE`, the allocation and deallocation routines are as follows:

```c
static TYPE_slice * TYPE_alloc(void);

This function allocates a `TYPE` array, returning a pointer to the allocated `TYPE` array. If the array cannot be dynamically allocated, 0 (zero) is returned.

static void TYPE_free(TYPE_slice * Value);

This function frees a dynamically allocated `TYPE` array. The `Value` argument is a pointer to the dynamically allocated `TYPE` array to be freed.
```
Exceptions

An exception in OMG IDL is mapped to a C++ class. The C++ class contains the following:

- Constructors
- Destructors
- A static \_narrow function, to determine the type of exception

The generated class is similar to a variable-length structure, but with an additional constructor to simplify initialization, and with the static \_narrow member function to determine the type of UserException.

For example, consider the following OMG IDL definition:

```idl
// OMG IDL
module INVENT
{
    exception NonExist
    {
        ID BadId;
    }
};
```

This definition maps to C++ as follows:

```cpp
// C++
class INVENT
{
    
    class NonExist : public CORBA::UserException
    {
        public:
            static NonExist * _narrow(CORBA::Exception_ptr);
            NonExist (ID _BadId);
            NonExist ();
            NonExist (const NonExist &);
            ~NonExist ();
            NonExist & operator=(const NonExist &);
            void _raise ();
    }
};
```
Attributes (data members) of the Exception class are public, so you may access them directly.

**Member Mapping**

Members of an exception are mapped in the same manner as structs. For more information, see **Member Mapping**.

All exception members are public data in the C++ class, and are accessed directly.

**Var**

A var class is generated for an exception. For more information, see the section **Using var Classes**.

**Out**

An out class is generated for an exception. For more information, see the section **Using out Classes**.

**Member Functions**

For a given OMG IDL exception *TYPE*, the generated member functions are as follows:

```
static TYPE * narrow(CORBA::Exception_ptr Except);

This function returns a pointer to a TYPE exception class if the exception can be narrowed to a TYPE exception. If the exception cannot be narrowed, 0 (zero) is returned. The TYPE pointer is not a pointer to a new class. Instead, it is a typed pointer to the original exception pointer and is valid only as long as the Except parameter is valid.

TYPE ( );

This is the default constructor for the exception. No initialization of members is performed for fixed-length members. Variable-length members map to self-managing types; these types have constructors that initialize the member.

TYPE (member-parameters);

This constructor has an argument for each of the members in the exception. The constructor copies each argument and does not assume ownership of the memory for any argument. Building on the previous example, the signature of the constructor is:

    NonExist (ID _BadId);
There is one argument for each member of the exception. The type and parameter-passing mechanism are identical to the Any insertion operator. For information about the Any insertion operator, see the section Insertion into Any.

```
TYPE (const TYPE & From);
```

This copy constructor copies the data from the specified TYPE exception argument. The From argument specifies the exception to be copied.

```
~TYPE ()
```

This destructor frees the data associated with the exception.

```
TYPE & operator=(const TYPE & From);
```

This assignment operator copies the data from the specified TYPE exception argument. The From argument specifies the exception to be copied.

```
void _raise ();
```

This function causes the exception instance to throw itself. A catch clause can catch it by a more derived type.

### Mapping of Pseudo-objects to C++

CORBA pseudo-objects may be implemented either as normal CORBA objects or as serverless objects. In the CORBA specification, the fundamental differences between these strategies are:

- Serverless object types do not inherit from CORBA::Object.
- Individual serverless objects are not registered with any ORB.
- Serverless objects do not necessarily follow the same memory management rules as for regular IDL types.

References to serverless objects are not necessarily valid across computational contexts; for example, address spaces. Instead, references to serverless objects that are passed as parameters may result in the construction of independent, functionally identical copies of objects used by receivers of these references. To support this, the otherwise hidden representational properties (such as data layout) of serverless objects are made known to the ORB. Specifications for achieving this are not contained in this chapter; making serverless objects known to the ORB is an implementation detail.

This chapter provides a standard mapping algorithm for all pseudo-object types. This avoids the need for piecemeal mappings for each of the nine CORBA pseudo-object types, and accommodates any pseudo-object types that may be proposed in future revisions of CORBA. It also avoids representation dependence in the C mapping, while still allowing implementations that rely on C-compatible representations.
Usage

Rather than C-PIDL, this mapping uses an augmented form of full OMG IDL to describe serverless object types. Interfaces for pseudo-object types follow the same rules as normal OMG IDL interfaces, with the following exceptions:

- They are prefaced by the keyword pseudo.
- Their declarations may refer to other serverless object types that are not otherwise necessarily allowed in OMG IDL.

The pseudo prefix means that the interface may be implemented in either a normal or serverless fashion. That is, apply either the rules described in the following sections, or the normal mapping rules described in this chapter.

Mapping Rules

Serverless objects are mapped in the same way as normal interfaces, except for the differences outlined in this section.

Classes representing serverless object types are not subclasses of CORBA::Object, and are not necessarily subclasses of any other C++ class. Thus, they do not necessarily support, for example, the Object::create_request operation.

For each class representing a serverless object type T, overloaded versions of the following functions are provided in the CORBA namespace:

```cpp
// C++
void release(T_ptr);
Boolean is_nil(T_ptr p);
```

The mapped C++ classes are not guaranteed to be usefully subclassable by users, although subclasses can be provided by implementations. Implementations are allowed to make assumptions about internal representations and transport formats that may not apply to subclasses.

The member functions of classes representing serverless object types do not necessarily obey the normal memory management rules. This is because some serverless objects, such as CORBA::NVList, are essentially just containers for several levels of other serverless objects. Requiring callers to explicitly free the values returned from accessor functions for the contained serverless objects would be counter to their intended usage.

---

1. In particular, exception used as a data type and a function name.
All other elements of the mapping are the same. In particular:

- The types of references to serverless objects, T_ptr, may or may not simply be a typedef of T*.

- Each mapped class supports the following static member functions:

  // C++
  static T_ptr _duplicate(T_ptr p);
  static T_ptr _nil();

- Legal implementations of _duplicate include simply returning the argument or constructing references to a new instance. Individual implementations may provide stronger guarantees about behavior.

- The corresponding C++ classes may or may not be directly instantiable or have other instantiation constraints. For portability, users should invoke the appropriate constructive operations.

- As with normal interfaces, assignment operators are not supported.

- Although they can transparently employ “copy-style” rather than “reference-style” mechanics, parameter passing signatures and rules as well as memory management rules are identical to those for normal objects, unless otherwise noted.

### Relation to the C PIDL Mapping

All serverless object interfaces and declarations that rely on them have direct analogs in the C mapping. The mapped C++ classes can, but need not, be implemented using representations compatible to those chosen for the C mapping. Differences between the pseudo-object specifications for C-PIDL and C++ PIDL are as follows:

- C++ PIDL calls for removal of representation dependencies through the use of interfaces rather than structs and typedefs.

- C++ PIDL calls for placement of operations on pseudo-objects in their interfaces, including a few cases of redesignated functionality as noted.

- In C++ PIDL, release performs the role of the associated free and delete operations in the C mapping, unless otherwise noted.

Brief descriptions and listings of each pseudo-interface and its C++ mapping are provided in the following sections. Further details, including definitions of types referenced but not defined below, may be found in the relevant sections of this document.
Typedefs

A typedef in OMG IDL is mapped to a typedef in C++. Depending upon the OMG IDL data type, additional typedefs and member functions may be defined. The generated code for each data type is as follows:

- **Basic data types (short, long, unsigned short, unsigned long, float, double, char, boolean, and octet)**

  Basic data types map to a simple typedef. For example:
  
  ```
  // OMG IDL
  typedef long ID;
  // C++
  typedef CORBA::Long ID;
  ```

- **string**

  A string typedef is mapped to a simple typedef. For example:
  
  ```
  // OMG IDL
  typedef string IDStr;
  // C++
  typedef char * IDStr;
  ```

- **object, interfaces, TypeCode**

  Object, interfaces, and TypeCode types are mapped to four typedefs. For example:
  
  ```
  // OMG IDL
  typedef Item Intf;
  // C++
  typedef Item Intf;
  typedef Item_ptr Intf_ptr;
  typedef Item_var Intf_var;
  typedef Item_ptr & Intf _out;
  ```

- **enum, struct, union, sequence**

  UDTs are mapped to three typedefs. For example:
  
  ```
  // OMG IDL
  typedef LogList ListRetType;
  // C++
  typedef LogList ListRetType;
  typedef LogList_var ListRetType_var;
  typedef LogList_out & ListRetType_out;
  ```
Arrays are mapped to four typedefs and the static member functions to allocate and free memory. For example:

```c++
// OMG IDL
typedef LogArray ArrayRetType;
// C++
typedef LogArray ArrayRetType;
typedef LogArray_var ArrayRetType_var;
typedef LogArray_forany ArrayRetType_forany;
typedef LogArray_slice ArrayRetType_slice;
ArrayRetType_slice * ArrayRetType_alloc();
void ArrayRetType_free(ArrayRetType_slice *);
```

### Implementing Interfaces

An operation in OMG IDL is mapped to a C++ member function.

The name of the member function is the name of the operation. The operation is defined as a member function in both the interface class and the stub class. The interface class is virtual; the stub class inherits from the virtual class and contains the member function code from the client application stub. When an operation is invoked on the object reference, the code contained in the corresponding stub member function executes.

For example, consider the following OMG IDL definition:

```omg
module INVENT
{
    interface Order
    {
        . . .
        ItemList modifyOrder (in ItemList ModifyList);
    };
};
```

This definition maps to C++ as follows:

```c++
// C++
class INVENT
{
    . . .
```
class Order : public virtual CORBA::Object
{
  . . .
  virtual ItemList * modifyOrder (const ItemList & ModifyList) = 0;
};

class Stub_Order : public Order
{
  . . .
  ItemList * modifyOrder (const ItemList & ModifyList);
};

The generated client application stub then contains the following generated code for the stub class:

// ROUTINE NAME: INVENT::Stub_Order::modifyOrder
// // FUNCTIONAL DESCRIPTION:
// // Client application stub routine for operation
// // modifyOrder. // (Interface : Order)
INVENT::ItemList * INVENT::Stub_Order::modifyOrder (const INVENT::ItemList & ModifyList)
{
  . . .
}

**Argument Mapping**

Each of the arguments in an operation is mapped to the corresponding C++ type as described in Table 13-1 and Table 13-2.

The parameter passing modes for arguments in an operation are described in Table 13-7 and Table 13-8.
Implementing Operations

The signature of an implementation member function is the mapped signature of the OMG IDL operation. Unlike the client side, the server-side mapping requires that the function header include the appropriate exception (throw) specification. This requirement allows the compiler to detect when an invalid exception is raised, which is necessary in the case of a local C++-to-C++ library call (otherwise, the call would have to go through a wrapper that checks for a valid exception). For example:

```idl
// IDL
interface A
{
exception B {};
void f() raises(B);
};
```

```cpp
// C++
class MyA : public virtual POA_A
{
public:
void f() throw(A::B, CORBA::SystemException);
...}
```

Since all operations and attributes may throw CORBA system exceptions, `CORBA::SystemException` must appear in all exception specifications, even when an operation has no raises clause.

Within a member function, the “this” pointer refers to the implementation object’s data as defined by the class. In addition to accessing the data, a member function may implicitly call another member function defined by the same class. For example:

```idl
// IDL
interface A
{
void f();
void g();
};
```
// C++
class MyA : public virtual POA_A
{
  public:
    void f() throw(SystemException);
    void g() throw(SystemException);
  private:
    long x_;
};

void
MyA::f() throw(SystemException)
{
  this->x_ = 3;
  this->g();
}

However, when a servant member function is invoked in this manner, it is being called simply as
a C++ member function, not as the implementation of an operation on a CORBA object. In such
a context, any information available via the POA_Current object refers to the CORBA request
invocation that performed the C++ member function invocation, not to the member function
invocation itself.

Skeleton Derivation from Object

In several existing ORB implementations, each skeleton class derives from the corresponding
interface class. For example, for interface Mod::A, the skeleton class POA_Mod::A is derived
from class Mod::A. These systems, therefore, allow an object reference for a servant to be
implicitly obtained via normal C++ derived-to-base conversion rules:

// C++
MyImplOfA my_a;    // declare impl of A
A_ptr a = &my_a;   // obtain its object reference
                   // by C++ derived-to-base conversion

Such code can be supported by a conforming ORB implementation, but it is not required, and is
thus not portable. The equivalent portable code invokes _this() on the implementation object
to implicitly register it if it has not yet been registered, and to get its object reference:
PortableServer Functions

Objects registered with POAs use sequences of octet, specifically the PortableServer::POA::ObjectId type, as object identifiers. However, because C++ programmers often want to use strings as object identifiers, the C++ mapping provides several conversion functions that convert strings to ObjectId and vice versa:

```cpp
// C++
namespace PortableServer
{
    char* ObjectId_to_string(const ObjectId&);
    ObjectId* string_to_ObjectId(const char*);
}
```

These functions follow the normal C++ mapping rules for parameter passing and memory management.

If conversion of an ObjectId to a string would result in illegal characters in the string (such as a NULL), the first two functions throw the CORBA::BAD_PARAM exception.

Modules

A module in OMG IDL is mapped to a C++ class. Objects contained in the module are defined within this C++ class. Because interfaces and types are also mapped to classes, nested C++ classes result.

For example, consider the following OMG IDL definition:

```idl
// OMG IDL
module INVENT
{
    interface Order
    {
        . . .
    };
};
```
This definition maps to C++ as follows:

```cpp
// C++

class INVENT
{
  ...
  class Order : public virtual CORBA::Object
  {
    ...
  }; // class Order
}; // class INVENT
```

Multiple nested modules yield multiple nested classes. Anything inside the module will be in the module class. Anything inside the interface will be in the interface class.

OMG IDL allows modules, interfaces, and types to have the same name. However, when generating files for the C++ language, having the same name is not allowed. This restriction is necessary because the OMG IDL names are generated into nested C++ classes with the same name; this is not supported by C++ compilers.

**Note:** The BEA Tuxedo OMG IDL compiler outputs an informational message if you generate C++ code from OMG IDL with an interface or type with the same name as the current module name. If you ignore this informational message and do not use unique names to differentiate the interface or type from the module name, the compiler will signal errors when compiling the generated files.

## Interfaces

An interface in OMG IDL is mapped to a C++ class. This class contains the definitions of the operations, attributes, constants, and user-defined types (UDTs) contained in the OMG IDL interface.

For an interface \textit{INTF}, the generated interface code contains the following items:

- Object reference type (\textit{INTF}\_ptr)
- Object reference variable type (\textit{INTF}\_var)
- \_duplicate static member function
- \_narrow static member function
- \_nil static member function
• UDTs
• Member functions for attributes and operations

For example, consider the following OMG IDL definition:

```
// OMG IDL
module INVENT
{
    interface Order
    {
        void cancelOrder ()
    }
};
```

This definition maps to C++ as follows:

```
// C++
class INVENT
{
    . . .
    class Order;
    typedef Order * Order_ptr;

    class Order : public virtual CORBA::Object
    {
        . . .
        static Order_ptr _duplicate(Order_ptr obj);
        static Order_ptr _narrow(CORBA::Object_ptr obj);
        static Order_ptr _nil();
        virtual void cancelOrder () = 0;
        . . .
    };
};
```

The object reference types and static member functions are described in the following sections, as are UDTs, operations, and attributes.

### Generated Static Member Functions

This section describes in detail the generated static member functions: _duplicate, _narrow, and _nil for an interface INTF.
static INTF_ptr _duplicate (INTF_ptr Obj)
This static member function duplicates an existing INTF object reference and returns a new INTF object reference. The new INTF object reference must be released by calling the CORBA::release member function. If an error occurs, a reference to the nil INTF object is returned. The argument Obj specifies the object reference to be duplicated.

static INTF_ptr _narrow (CORBA::Object_ptr Obj)
This static member function returns a new INTF object reference given an existing CORBA::Object_ptr object reference. The Object_ptr object reference may have been created by a call to the CORBA::ORB::string_to_object member function or may have been returned as a parameter from an operation.

The INTF_ptr object reference must correspond to an INTF object or to an object that inherits from the INTF object. The new INTF object reference must be released by calling the CORBA::release member function. The argument Obj specifies the object reference to be narrowed to an INTF object reference. The Obj parameter is not modified by this member function and should be released by the user when it is no longer required. If Obj cannot be narrowed to an INTF object reference, the INTF nil object reference is returned.

static INTF_ptr _nil ( )
This static member function returns the new nil object reference for the INTF interface. The new reference does not have to be released by calling the CORBA::release member function.

Object Reference Types
An interface class (INTF) is a virtual class; the CORBA standard does not allow you to:

- Create or hold an instance of the interface class
- Use a pointer or a reference to the interface class

Instead, you use one of the object reference types, INTF_ptr or INTF_var class. You can obtain an object reference by using the _narrow static member function. Operations are invoked on these classes using the arrow operator (->).

The INTF_var class simplifies memory management by automatically releasing the object reference when the INTF_var class goes out of scope or is reassigned. Variable types are generated for many of the UDTs and are described in Using var Classes.

Attributes
A read-only attribute in OMG IDL is mapped to a C++ function that returns the attribute value. A read-write attribute maps to two overloaded C++ functions, one to return the attribute value
and one to set the attribute value. The name of the overloaded member function is the name of the attribute.

Attributes are generated in the same way that operations are generated. They are defined in both the virtual and the stub classes. For example, consider the following OMG IDL definition:

```idl
// OMG IDL
module INVENT
{
    interface Order
    {
        . . .
        attribute itemStruct itemInfo;
    };
};
```

This definition maps to C++ as follows:

```cpp
// C++
class INVENT
{
    . . .

class Item : public virtual CORBA::Object
    {
        . . .
        virtual itemStruct * itemInfo () = 0;
        virtual void itemInfo (
            const itemStruct & itemInfo) = 0;
    };
};
class Stub_Item : public Item
{
    . . .
    itemStruct * itemInfo ();
    void itemInfo (
        const itemStruct & itemInfo);
};
```
The generated client application stub then contains the following generated code for the stub class:

```c
// ROUTINE NAME:       INVENT::Stub_Item::itemInfo
//
// FUNCTIONAL DESCRIPTION:
//
//   Client application stub routine for attribute
//   INVENT::Stub_Item::itemInfo. (Interface : Item)

INVENT::itemStruct * INVENT::Stub_Item::itemInfo ( )
{
 . . .
}

// ROUTINE NAME:       INVENT::Stub_Item::itemInfo
//
// FUNCTIONAL DESCRIPTION:
//
//   Client application stub routine for attribute
//   INVENT::Stub_Item::itemInfo. (Interface : Item)

void INVENT::Stub_Item::itemInfo ( const INVENT::itemStruct & itemInfo )
{
}
```

**Argument Mapping**

An attribute is equivalent to two operations, one to return the attribute and one to set the attribute. For example, the `itemInfo` attribute listed above is equivalent to:

```c
void itemInfo (in itemStruct itemInfo);
itemStruct itemInfo ();
```

The argument mapping for the attribute is the same as the mapping for an operation argument. The attribute is mapped to the corresponding C++ type as described in Table 13-1 and Table 13-2. The parameter passing modes for arguments in an operation are described in Table 13-7 and Table 13-8.
Any Type

An any in OMG IDL is mapped to the CORBA::Any class. The CORBA::Any class handles C++ types in a type-safe manner.

Handling Typed Values

To decrease the chances of creating an any with a mismatched TypeCode and value, the C++ function overloading facility is utilized. Specifically, for each distinct type in an OMG IDL specification, overloaded functions to insert and extract values of that type are provided. Overloaded operators are used for these functions to completely avoid any name space pollution. The nature of these functions, which are described in detail below, is that the appropriate TypeCode is implied by the C++ type of the value being inserted into or extracted from the any.

Since the type-safe any interface described below is based upon C++ function overloading, it requires C++ types generated from OMG IDL specifications to be distinct. However, there are special cases in which this requirement is not met:

- The Boolean, octet, and char OMG IDL types are not required to map to distinct C++ types, which means that a separate means of distinguishing them from each other for the purpose of function overloading is necessary. The means of distinguishing these types from each other is described in Distinguishing Boolean, Octet, Char, and Bounded Strings.

- Since all strings are mapped to char* regardless of whether they are bounded or unbounded, another means of creating or setting an any with a bounded string value is necessary. This is described in Distinguishing Boolean, Octet, Char, and Bounded Strings.

- In C++, arrays within a function argument list decay into pointers to their first elements. This means that function overloading cannot be used to distinguish between arrays of different sizes. The means for creating or setting an any when dealing with arrays is described below and in Arrays.

Insertion into Any

To allow a value to be set in an any in a type-safe fashion, the following overloaded operator function is provided for each separate OMG IDL type T:

```cpp
// C++
void operator<<(Any&, T);
```

This function signature suffices for the following types, which are usually passed by value:

- Short, UShort, Long, ULONG, Float, Double
Enumerations

Unbounded strings (char* passed by value)

Object references (T_ptr)

For values of type T that are too large to be passed by value efficiently, two forms of the insertion function are provided:

```cpp
// C++
void operator<<=(Any&, const T&);     // copying form
void operator<<=(Any&, T*);            // non-copying form
```

Note that the copying form is largely equivalent to the first form shown, as far as the caller is concerned.

These “left-shift-assign” operators are used to insert a typed value into an any, as follows:

```cpp
// C++
Long value = 42;
Any a;
a <<= value;
```

In this case, the version of operator<<= overloaded for type Long sets both the value and the TypeCode properly for the Any variable.

Setting a value in an any using operator<<= means the following:

- For the copying version of operator<<=, the lifetime of the value in the Any is independent of the lifetime of the value passed to operator<<=. The implementation of the Any does not store its value as a reference or a pointer to the value passed to operator<<=.

- For the noncopying version of operator<<=, the inserted T* is consumed by the Any. The caller may not use the T* to access the pointed-to data after insertion because the Any assumes ownership of T*, and the Any may immediately copy the pointed-to data and destroy the original.

- With both the copying and noncopying versions of operator<<=, any previous value held by the Any is properly deallocated. For example, if the Any(TypeCode_ptr,void*,TRUE) constructor (described in Handling Untyped Values) were called to create the Any, the Any is responsible for deallocating the memory pointed to by the void* before copying the new value.

Copying insertion of a string type causes the following function to be invoked:
Since all string types are mapped to `char*`, this insertion function assumes that the value being inserted is an unbounded string. Distinguishing Boolean, Octet, Char, and Bounded Strings describes how bounded strings may be correctly inserted into an `Any`. Noncopying insertion of both bounded and unbounded strings can be achieved using the `Any::from_string` helper type described in Distinguishing Boolean, Octet, Char, and Bounded Strings.

Type-safe insertion of arrays uses the `Array_forany` types described in Arrays. The ORB provides a version of `operator<<="` overloaded for each `Array_forany` type. For example:

```
// IDL
typedef long LongArray[4][5];
// C++
typedef Long LongArray[4][5];
typedef Long LongArray_slice[5];
class LongArray_forany { ... };
void operator<<=(Any &, const LongArray_forany &);
```

The `Array_forany` types are always passed to `operator<<="` by reference to `const`. The `nocopy` flag in the `Array_forany` constructor is used to control whether the inserted value is copied (`nocopy == FALSE`) or consumed (`nocopy == TRUE`). Because the `nocopy` flag defaults to `FALSE`, copying insertion is the default.

Because of the type ambiguity between an array of `T` and a `T*`, it is highly recommended that portable code explicitly use the appropriate `Array_forany` type when inserting an array into an `Any`. For example:

```
// IDL
struct S {... };
typedef S SA[5];
// C++
struct S { ... };
typedef S SA[5];
typedef S SA_slice;
class SA_forany { ... };
SA s;
```
// ...initialize s...
Any a;
a <<= s;       // line 1
a <<= SA_forany(s);   // line 2

Line 1 results in the invocation of the noncopying operator<<(Any&, S*) due to the decay of the SA array type into a pointer to its first element, rather than the invocation of the copying SA_forany insertion operator. Line 2 explicitly constructs the SA_forany type and thus results in the desired insertion operator being invoked.

The noncopying version of operator<< for object references takes the address of the T_ptr type, as follows:

// IDL
interface T { ... };

// C++
void operator<<(Any&, T_ptr);       // copying
void operator<<(Any&, T_ptr*);      // non-copying

The noncopying object reference insertion consumes the object reference pointed to by T_ptr*; therefore, after insertion the caller may not access the object referred to by T_ptr because the Any may have duplicated and then immediately released the original object reference. The caller maintains ownership of the storage for the T_ptr itself.

The copying version of operator<< is also supported on the Any_var type.

**Extraction from Any**

To allow type-safe retrieval of a value from an any, the ORB provides the following operators for each OMG IDL type T:

// C++
Boolean operator>>(const Any&, T&);

This function signature suffices for primitive types that are usually passed by value. For values of type T that are too large to be passed by value efficiently, the ORB provides a different signature, as follows:

// C++
Boolean operator>>(const Any&, T*);

The first form of this function is used only for the following types:

- Boolean, Char, Octet, Short, UShort, Long, ULong, Float, Double
Mappings

- Enumerations
- Unbounded strings (char* passed by reference, i.e., char*s)
- Object references (T_ptr)

For all other types, the second form of the function is used.

This “right-shift-assign” operator is used to extract a typed value from an any, as follows:

```cpp
// C++
Long value;
Any a;
a <<= Long(42);
if (a >>= value) {
    // ... use the value ...
}
```

In this case, the version of operator>>= for type Long determines whether the Any truly does contain a value of type Long and, if so, copies its value into the reference variable provided by the caller and returns TRUE. If the Any does not contain a value of type Long, the value of the caller’s reference variable is not changed, and operator>>= returns FALSE.

For nonprimitive types, extraction is done by pointer. For example, consider the following OMG IDL struct:

```idl
struct MyStruct {
    long lmem;
    short smem;
};
```

Such a struct could be extracted from an Any as follows:

```cpp
// C++
Any a;
// ... a is somehow given a value of type MyStruct ...
MyStruct *struct_ptr;
if (a >>= struct_ptr) {
    // ... use the value ...
}
```

If the extraction is successful, the caller’s pointer points to storage managed by the Any, and operator>>= returns TRUE. The caller must not try to delete or otherwise release this storage.
The caller also should not use the storage after the contents of the Any variable are replaced via assignment, insertion, or the replace function, or after the Any variable is destroyed. Care must be taken to avoid using T_var types with these extraction operators, since they will try to assume responsibility for deleting the storage owned by the Any.

If the extraction is not successful, the value of the caller’s pointer is set equal to the NULL pointer, and operator>>= returns FALSE.

Correct extraction of array types relies on the Array_forany types described in Arrays.

An example of the OMG IDL is as follows:

// IDL
typedef long A[20];
typedef A B[30][40][50];

// C++
typedef Long A[20];
typedef Long A_slice;
class A_forany { ... };
typedef A B[30][40][50];
typedef A B_slice[40][50];
class B_forany { ... };

Boolean operator>>=(const Any&, A_forany&); // for type A
Boolean operator>>=(const Any&, B_forany&); // for type B

The Array_forany types are always passed to operator>>= by reference.

For strings and arrays, applications are responsible for checking the TypeCode of the Any to be sure that they do not overstep the bounds of the array or string object when using the extracted value.

The operator>>= is also supported on the Any_var type.

**Distinguishing Boolean, Octet, Char, and Bounded Strings**

Since the Boolean, octet, and char OMG IDL types are not required to map to distinct C++ types, another means of distinguishing them from each other is necessary so that they can be used with the type-safe Any interface. Similarly, since both bounded and unbounded strings map to char*,
another means of distinguishing them must be provided. This is done by introducing several new helper types nested in the Any class interface. For example, this is accomplished as shown below:

```cpp
// C++
class Any
{
    public:
        // special helper types needed for boolean, octet,
        // char, and bounded string insertion
        struct from_boolean {
            from_boolean(Boolean b) : val(b) {}
            Boolean val;
        };
        struct from_octet {
            from_octet(Octet o) : val(o) {}
            Octet val;
        };
        struct from_char {
            from_char(Char c) : val(c) {}
            Char val;
        };
        struct from_string {
            from_string(char* s, ULong b,
                        Boolean nocopy = FALSE) :
            val(s), bound(b) {}
            char *val;
            ULong bound;
        };
        void operator<<(from_boolean);
        void operator<<(from_char);
        void operator<<(from_octet);
        void operator<<(from_string);
        // special helper types needed for boolean, octet,
        // char, and bounded string extraction
        struct to_boolean {
            to_boolean(Boolean &b) : ref(b) {}
            Boolean &ref;
        };
```
struct to_char {
    to_char(Char &c) : ref(c) {} 
    Char &ref;
};
struct to_octet {
    to_octet(Octet &o) : ref(o) {} 
    Octet &ref;
};
struct to_string {
    to_string(char *&s, ULong b) : val(s), bound(b) {} 
    char *&val;
    ULong bound;
};
Boolean operator>>=(to_boolean) const;
Boolean operator>>=(to_char) const;
Boolean operator>>=(to_octet) const;
Boolean operator>>=(to_string) const;
// other public Any details omitted
private:
    // these functions are private and not implemented
    // hiding these causes compile-time errors for
    // unsigned char
    void operator<<=(unsigned char);
    Boolean operator>=(unsigned char & const);
};

The ORB provides the overloaded operator<<= and operator>= functions for these special helper types. These helper types are used as shown here:

// C++
Boolean b = TRUE;
Any any;
any <<= Any::from_boolean(b);
// ...
if (any >>= Any::to_boolean(b)) {
    // ...any contained a Boolean...
char* p = "bounded";
any <<= Any::from_string(p, 8);
// ...
if (any >>= Any::to_string(p, 8)) { 
    // ...any contained a string<8>...
}

A bound value of 0 (zero) indicates an unbounded string.

For noncopying insertion of a bounded or unbounded string into an Any, the nocopy flag on the from_string constructor should be set to TRUE:

// C++
char* p = string_alloc(8);
// ...initialize string p...
any <<= Any::from_string(p, 8, 1); // any consumes p

Assuming that boolean, char, and octet all map the C++ type unsigned char, the private and unimplemented operator<<= and operator>>= functions for unsigned char cause a compile-time error if straight insertion or extraction of any of the boolean, char, or octet types is attempted:

// C++
Octet oct = 040;
Any any;
any <<= oct; // this line will not compile
any <<= Any::from_octet(oct); // but this one will

**Widening to Object**

Sometimes it is desirable to extract an object reference from an Any as the base Object type. This can be accomplished using a helper type similar to those required for extracting boolean, char, and octet:

// C++
class Any {
  public:
    ...
    struct to_object {
      to_object(Object_ptr &obj) : ref(obj) {} 
      Object_ptr &ref;
    }
The `to_object` helper type is used to extract an object reference from an Any as the base Object type. If the Any contains a value of an object reference type as indicated by its TypeCode, the extraction function `operator>>(to_object)` explicitly widens its contained object reference to Object and returns `TRUE`; otherwise, it returns `FALSE`. This is the only object reference extraction function that performs widening on the extracted object reference. As with regular object reference extraction, no duplication of the object reference is performed by the `to_object` extraction operator.

**Handling Untyped Values**

Under some circumstances the type-safe interface to Any is not sufficient. An example is a situation in which data types are read from a file in binary form and are used to create values of type Any. For these cases, the Any class provides a constructor with an explicit TypeCode and generic pointer:

```cpp
// C++
Any(TypeCode_ptr tc, void *value, Boolean release = FALSE);
```

The constructor duplicates the given TypeCode pseudo-object reference. If the `release` parameter is `TRUE`, the Any object assumes ownership of the storage pointed to by the `value` parameter. A caller should make no assumptions about the continued lifetime of the `value` parameter once it has been handed to an Any with `release=TRUE`, since the Any may copy the `value` parameter and immediately free the original pointer. If the `release` parameter is `FALSE` (the default case), the Any object assumes that the caller manages the memory pointed to by `value`. The `value` parameter can be a NULL pointer.

The Any class also defines three unsafe operations:

```cpp
// C++
void replace(
    TypeCode_ptr,
    void *value,
    Boolean release = FALSE
);
TypeCode_ptr type() const;
const void *value() const;
```
The replace function is intended to be used with types that cannot be used with the type-safe insertion interface, and so is similar to the constructor described above. The existing TypeCode is released and value storage is deallocated, if necessary. The TypeCode function parameter is duplicated. If the release parameter is TRUE, the Any object assumes ownership for the storage pointed to by the value parameter. The Any should make no assumptions about the continued lifetime of the value parameter once it has been handed to the Any::replace function with release=TRUE, since the Any may copy the value parameter and immediately free the original pointer. If the release parameter is FALSE (the default case), the Any object assumes that the caller manages the memory occupied by the value. The value parameter of the replace function can be a NULL pointer.

Note that neither the constructor shown above nor the replace function is type-safe. In particular, no guarantees are made by the compiler at run time as to the consistency between the TypeCode and the actual type of the void* argument. The behavior of an ORB implementation when presented with an Any that is constructed with a mismatched TypeCode and value is not defined.

The type function returns a TypeCode_ptr pseudo-object reference to the TypeCode associated with the Any. Like all object reference return values, the caller must release the reference when it is no longer needed, or assign it to a TypeCode_var variable for automatic management.

The value function returns a pointer to the data stored in the Any. If the Any has no associated value, the value function returns a NULL pointer.

**Any Constructors, Destructor, Assignment Operator**

The default constructor creates an Any with a TypeCode of type tk_null, and no value. The copy constructor calls _duplicate on the TypeCode_ptr of its Any parameter and deep-copies the parameter’s value. The assignment operator releases its own TypeCode_ptr and deallocates storage for the current value if necessary, then duplicates the TypeCode_ptr of its Any parameter and deep-copies the parameter’s value. The destructor calls release on the TypeCode_ptr and deallocates storage for the value, if necessary.

Other constructors are described in the section Handling Untyped Values.

**The Any Class**

The full definition of the Any class can be found in the section Any Class Member Functions.
Value Type


Overview

Objects, more specifically, interface types that objects support, are defined in an IDL interface, allowing arbitrary implementations. There is great value in having a distributed object system that places almost no constraints on implementation. However, there are many occasions in which it is desirable to be able to pass an object by value, rather than by reference. This may be particularly useful when an object’s primary “purpose” is to encapsulate data, or an application explicitly wishes to make a “copy” of an object.

The semantics of passing an object by value are similar to that of standard programming languages. The receiving side of a parameter passed by value receives a description of the “state” of the object. It then instantiates a new instance with that state but having a separate identity from that of the sending side. Once the parameter passing operation is complete, no relationship is assumed to exist between the two instances.

Because it is necessary for the receiving side to instantiate an instance, it must necessarily know something about the object’s state and implementation. Thus, valuetype(s) provide semantics that bridge between CORBA structs and CORBA interfaces, as follows:

- They support description of complex state (that is, arbitrary graphs, with recursion and cycles).
- Their instances are always local to the context in which they are used (because they are always copied when passed as a parameter to a remote call).
- They support both public and private (to the implementation) data members.
- They can be used to specify the state of an object implementation (that is, they can support an interface).
- They support single inheritance (of valuetype) and can support an interface.
- They may be also be abstract.
Architecture

The basic notion of valuetypes is relatively simple. A valuetype is, in some sense, half way between a “regular” IDL interface type and a struct. The use of valuetype is a signal from the application programmer that some additional properties (state) and implementation details be specified beyond that of an interface type. Specification of this information puts some additional constraints on the implementation choices beyond that of interface types. This is reflected in both the semantics specified herein, and in the language mappings.

Benefits

Prior to supporting valuetypes (objects passable by value), all CORBA objects had object references. When multiple clients invoked on a particular object, they use the same object reference. The instance(s) of the object remained on the server ORB and its state was maintained by the server ORB, not the client ORB.

Valuetypes represent a significant addition to the CORBA architecture. As with objects passed by reference, valuetypes have state and methods, but do not have object references and are always invoked locally as programming language objects. Upon request from the receiving side, valuetypes package their state in the sending context, send their state “over the wire” to the receiving side, where an instance is created and populated with the transmitted state. The sending side has no further control of the client-side instance. Thus, the receiving side can make subsequent invocations of the instance locally. This model eliminates the delays involved when communicating over the network. These delays can be significant in large networks. The addition of valuetypes enables CORBA implementations to more easily scale to meet large data-handling requirements.

Therefore, an essential property of valuetypes is that their implementations are always local. That is, the explicit use of valuetypes in a concrete programming language is always guaranteed to use a local implementation, and will not require a remote call. They have no identity (their value is their identity) and they are not “registered” with the ORB.

Valuetype Example

For example, consider the following IDL valuetype taken from the CORBA C++ Language Mapping Specification, June 1999, published by the Object Management Group (OMG):

```idl
// IDL
datatype Example {
    short op1();
    long op2(in Example x);
}
```
private short val1;
public long val2;
private string val3;
private float val4;
private Example val5;
};
The C++ mapping for this valuetype is:

// C++
class Example : public virtual ValueBase {
public:
    virtual Short op1() = 0;
    virtual Long op2(Example*) = 0;
    virtual Long val2() const = 0;
    virtual void val2(Long) = 0;
    static Example* _downcast(ValueBase*);
protected:
    Example();
    virtual ~Example();
    virtual Short val1() const = 0;
    virtual void val1(Short) = 0;
    virtual const char* val3() const = 0;
    virtual void val3(char*) = 0;
    virtual void val3(const char*) = 0;
    virtual void val3(const String_var&) = 0;
    virtual Float val4() const = 0;
    virtual void val4(Float) = 0;
    virtual Example* val5() const = 0;
    virtual void val5(Example*) = 0;

private:
    // private and unimplemented
    void operator=(const Example&);
Fixed-length Versus Variable-length User-defined Types

The memory management rules and member function signatures for a user-defined type depend upon whether the type is fixed-length or variable-length. A user-defined type is variable-length if it is one of the following:

- A bounded or unbounded string
- A bounded or unbounded sequence
- A struct or union that contains a variable-length member
- An array with a variable-length element type
- A typedef to a variable-length type

class OBV_Example : public virtual Example {
    public:
        virtual Long val2() const;
        virtual void val2(Long);

    protected:
        OBV_Example();
        OBV_Example(Short init_val1, Long init_val2,
                    const char* init_val3, Float init_val4,
                    Example* init_val5);
        virtual ~OBV_Example();
        virtual Short val1() const;
        virtual void val1(Short);
        virtual const char* val3() const;
        virtual void val3(char*);
        virtual void val3(const char*);
        virtual void val3(const String_var&);
        virtual Float val4() const;
        virtual void val4(Float);
        virtual Example* val5() const;
        virtual void val5(Example*);
        // ...
};
If a type is not on this list, the type is fixed-length.

**Using var Classes**

Automatic variables (vars) are provided to simplify memory management. Vars are provided through a var class that assumes ownership for the memory required for the type and frees the memory when the instance of the var object is destroyed or when a new value is assigned to the var object.

The BEA Tuxedo provides var classes for the following types:

- **String** (`CORBA::String_var`)
- **Object references** (`CORBA::Object_var`)
- **User-defined OMG IDL types** (`struct, union, sequence, array, and interface`)

The var classes have common member functions, but may support additional operators depending upon the OMG IDL type. For an OMG IDL type `TYPE`, the `TYPE_var` class contains constructors, destructors, assignment operators, and operators to access the underlying `TYPE` type. An example var class is as follows:

```cpp
class TYPE_var
{
public:
  // constructors
  TYPE_var();
  TYPE_var(TYPE *);
  TYPE_var(const TYPE_var &);
  // destructor
  ~TYPE_var();

  // assignment operators
  TYPE_var &operator=(TYPE *);
  TYPE_var &operator=(const TYPE_var &);

  // accessor operators
  TYPE *operator->();
  TYPE *operator->() const;
```
TYPE_var_ptr in() const;
TYPE_var_ptr& inout();
TYPE_var_ptr& out();

TYPE_var_ptr _retn();
operator const TYPE_ptr&() const;
operator TYPE_ptr&();
operator TYPE_ptr;

The details of the member functions are as follows:

**TYPE_var()**
This is the default constructor for the `TYPE_var` class. The constructor initializes to 0 (zero) the `TYPE` * owned by the var class. You may not invoke the `operator->` on a `TYPE_var` class unless a valid `TYPE` * has been assigned to it.

**TYPE_var(TYPE * Value);**
This constructor assumes ownership of the specified `TYPE` * parameter. When the `TYPE_var` is destroyed, the `TYPE` is released. The `Value` argument is a pointer to the `TYPE` to be owned by this var class. This pointer must not be 0 (zero).

**TYPE_var(const TYPE_var & From);**
This copy constructor allocates a new `TYPE` and makes a deep copy of the data contained in the `TYPE` owned by the `From` parameter. When the `TYPE_var` is destroyed, the copy of the `TYPE` is released or deleted. The `From` parameter specifies the var class that points to the `TYPE` to be copied.

**~TYPE_var();**
This destructor uses the appropriate mechanism to release the `TYPE` owned by the var class. For strings, this is the `CORBA::string_free` routine. For object references, this is the `CORBA::release` routine. For other types, this may be `delete` or a generated static routine used to free allocated memory.

**TYPE_var &operator=(TYPE * NewValue);**
This assignment operator assumes ownership of the `TYPE` pointed to by the `NewValue` parameter. If the `TYPE_var` currently owns a `TYPE`, it is released before assuming ownership of the `NewValue` parameter. The `NewValue` argument is a pointer to the `TYPE` to be owned by this var class. This pointer must not be 0 (zero).

**TYPE_var &operator=(const TYPE_var &From);**
This assignment operator allocates a new `TYPE` and makes a deep copy of the data contained in the `TYPE` owned by the `From` `TYPE_var` parameter. If `TYPE_var` currently
owns a TYPE, it is released. When the TYPE_var is destroyed, the copy of the TYPE is released. The From parameter specifies the var class that points to the data to be copied.

```cpp
TYPE *operator->();
TYPE *operator->() const;
```

These operators return a pointer to the TYPE owned by the var class. The var class continues to own the TYPE and it is the responsibility of the var class to release TYPE. You cannot use the operator-> until the var owns a valid TYPE. Do not try to release this return value or access this return value after the TYPE_var has been destroyed.

```cpp
TYPE_var_ptr in() const;
TYPE_var_ptr& inout();
TYPE_var_ptr& out();
TYPE_var_ptr _retn();
```

Because implicit conversions can sometimes cause a problem with some C++ compilers and with code readability, the TYPE_var types also support member functions that allow them to be explicitly converted for purposes of parameter passing. To pass a TYPE_var and an in parameter, call the in() member function; for inout parameters, the inout() member function; for out parameters, the out() member function. To obtain a return value from the TYPE_var, call the _return() function. For each TYPE_var type, the return types of each of these functions will match the type shown in Table 13-7 for the in, inout, out, and return modes for the underlying type TYPE, respectively.

Some differences occur in the operators supported for the user-defined data types. Table 13-3 describes the various operators supported by each OMG IDL data type, in the generated C++ code. Because the assignment operators are supported for all of the data types described in Table 13-3, they are not included in the comparison.

### Table 13-3 Comparison of Operators Supported for User-defined Data Type var Classes

<table>
<thead>
<tr>
<th>OMG IDL Data Type</th>
<th>operator -&gt;</th>
<th>operator[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>struct</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>union</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>sequence</td>
<td>Yes</td>
<td>Yes, non-const only</td>
</tr>
<tr>
<td>array</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

The signatures are as shown in Table 13-4.
### Sequence vars

Sequence vars support the following additional `operator[]` member function:

```c++
TYPE &operator[](CORBA::ULong Index);
```

This operator invokes the `operator[]` of sequence owned by the var class. The `operator[]` returns a reference to the appropriate element of the sequence at the specified index. The `Index` argument specifies the index of the element to return. This index cannot be greater than the current sequence length.

### Array vars

Array vars do not support `operator->`, but do support the following additional `operator[]` member functions to access the array elements:

```c++
TYPE_slice & operator[](CORBA::ULong Index);
const TYPE_slice & operator[](CORBA::ULong Index) const;
```

These operators return a reference to the array slice at the specified index. An array slice is an array with all the dimensions of the original array except the first dimension. The member functions for the array-generated classes use a pointer to a slice to return pointers to an array. The `Index` argument specifies the index of the slice to return. This index cannot be greater than the array dimension.

---

**Table 13-4 Operator Signatures for _var Classes**

<table>
<thead>
<tr>
<th>OMG IDL Data Type</th>
<th>Operator Member Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>struct</td>
<td>TYPE * operator-&gt; ()</td>
</tr>
<tr>
<td></td>
<td>TYPE * operator-&gt; () const</td>
</tr>
<tr>
<td>union</td>
<td>TYPE * operator-&gt; ()</td>
</tr>
<tr>
<td></td>
<td>TYPE * operator-&gt; () const</td>
</tr>
<tr>
<td>sequence</td>
<td>TYPE * operator-&gt; ()</td>
</tr>
<tr>
<td></td>
<td>TYPE * operator-&gt; () const</td>
</tr>
<tr>
<td></td>
<td>TYPE &amp; operator[](CORBA::Long index)</td>
</tr>
<tr>
<td>array</td>
<td>TYPE_slice &amp; operator[](CORBA::Long index)</td>
</tr>
<tr>
<td></td>
<td>TYPE_slice &amp; operator[](CORBA::Long index) const</td>
</tr>
</tbody>
</table>
String vars

The String vars in the member functions described in this section and in the section Sequence vars have a TYPE of char *. String vars support additional member functions, as follows:

String_var(char * str)
This constructor makes a String_var from a string. The str argument specifies the string that will be assumed. The user must not use the str pointer to access data.

String_var(const char * str)
String_var(const String_var & var)
This constructor makes a String_var from a const string. The str argument specifies the const string that will be copied. The var argument specifies a reference to the string to be copied.

String_var & operator=(char * str)
This assignment operator first releases the contained string using CORBA::string_free, and then assumes ownership of the input string. The str argument specifies the string whose ownership will be assumed by this String_var object.

String_var & operator=(const char * str)
String_var & operator=(const String_var & var)
This assignment operator first releases the contained string using CORBA::string_free, and then copies the input string. The Data argument specifies the string whose ownership will be assumed by this String_var object.

char operator[](Ulong Index)
char operator[](Ulong Index) const
These array operators are superscripting operators that provide access to characters within the string. The Index argument specifies the index of the array to use in accessing a particular character within the array. Zero-based indexing is used. The returned value of the Char operator[] (Ulong Index) function can be used as an lvalue. The returned value of the Char operator[] (Ulong Index) const function cannot be used as an lvalue.

out Classes

Structured types (struct, union, sequence), arrays, and interfaces have a corresponding generated _out class. The out class is provided for simplifying the memory management of pointers to variable-length and fixed-length types. For more information about out classes and the common member functions, see the section Using out Classes.

Some differences occur in the operators supported for the user-defined data types. Table 13-5 describes the various operators supported by each OMG IDL data type, in the generated C++
code. Because the assignment operators are supported for all of the data types described in Table 13-3, they are not included in the comparison.

The signatures are as shown in Table 13-6.

Table 13-6  Operator Signatures for _out Classes

<table>
<thead>
<tr>
<th>OMG IDL Data Type</th>
<th>Operator Member Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>struct</td>
<td>TYPE * operator-&gt; ()</td>
</tr>
<tr>
<td></td>
<td>TYPE * operator-&gt; () const</td>
</tr>
<tr>
<td>union</td>
<td>TYPE * operator-&gt; ()</td>
</tr>
<tr>
<td></td>
<td>TYPE * operator-&gt; () const</td>
</tr>
<tr>
<td>sequence</td>
<td>TYPE * operator-&gt; ()</td>
</tr>
<tr>
<td></td>
<td>TYPE * operator-&gt; () const</td>
</tr>
<tr>
<td></td>
<td>TYPE &amp; operator[](CORBA::Long index)</td>
</tr>
<tr>
<td>array</td>
<td>TYPE_slice &amp; operator[](CORBA::Long index)</td>
</tr>
<tr>
<td></td>
<td>TYPE_slice &amp; operator[](CORBA::Long index) const</td>
</tr>
</tbody>
</table>

Using out Classes

When a TYPE_var is passed as an out parameter, any previous value it referred to must be implicitly deleted. To give the ORB enough hooks to meet this requirement, each T_var type has a corresponding TYPE_out type that is used solely as the out parameter type.

Note: The _out classes are not intended to be instantiated directly by the programmer. Specify an _out class only in function signatures.
The general form for `TYPE_out` types for variable-length types is as follows:

```cpp
// C++
class TYPE_out
{
public:
    TYPE_out(TYPE*& p) : ptr_(p) { ptr_ = 0; }
    TYPE_out(TYPE_var& p) : ptr_(p.ptr_) { delete ptr_; ptr_ = 0; }
    TYPE_out(TYPE_out& p) : ptr_(p.ptr_) {};
    TYPE_out& operator=(TYPE_out& p) { ptr_ = p.ptr_; 
        return *this;
    }
    operator Type_out*() { return ptr_; }
    Type_out* ptr() { return ptr_; }

private:
    Type_out* ptr_;
    // assignment from TYPE_var not allowed
    void operator=(const TYPE_var&);
};
```

The first constructor binds the reference data member with the `T*` argument and sets the pointer to the zero (0) pointer value. The second constructor binds the reference data member with the pointer held by the `TYPE_var` argument, and then calls `delete` on the pointer (or `string_free()` in the case of the `String_out` type or `TYPE_free()` in the case of a `TYPE_var` for an array type `TYPE`). The third constructor, the copy constructor, binds the reference data member to the same pointer referenced by the data member of the constructor argument.

Assignment from another `TYPE_out` copies the `TYPE*` referenced by the `TYPE_out` argument to the data member. The overloaded assignment operator for `TYPE*` simply assigns the pointer argument to the data member. Note that assignment does not cause any previously held pointer to be deleted; in this regard, the `TYPE_out` type behaves exactly as a `TYPE*`. The `TYPE*` conversion operator returns the data member. The `ptr()` member function, which can be used to avoid having to rely on implicit conversion, also returns the data member. The overloaded arrow operator (`operator->()`) allows access to members of the data structure pointed to by the `TYPE*`
data member. Compliant applications may not call the overloaded `operator->()` unless the `TYPE_out` has been initialized with a valid nonNULL `TYPE*`.

Assignment to a `TYPE_out` from instances of the corresponding `TYPE_var` type is disallowed because there is no way to determine whether the application developer wants a copy to be performed, or whether the `TYPE_var` should yield ownership of its managed pointer so it can be assigned to the `TYPE_out`. To perform a copy of a `TYPE_var` to a `TYPE_out`, the application should use `new`, as follows:

```c++
// C++
TYPE_var t = ...;
my_out = new TYPE(t.in()); // heap-allocate a copy
```

The `in()` function called on `t` typically returns a `const TYPE&`, suitable for invoking the copy constructor of the newly allocated `T` instance.

Alternatively, to make the `TYPE_var` yield ownership of its managed pointer so it can be returned in a `T_out` parameter, the application should use the `TYPE_var::_retn()` function, as follows:

```c++
// C++
TYPE_var t = ...;
my_out = t._retn();       // t yields ownership, no copy
```

Note that the `TYPE_out` types are not intended to serve as general-purpose data types to be created and destroyed by applications; they are used only as types within operation signatures to allow necessary memory management side-effects to occur properly.

## Object Reference out Parameter

When a `_var` is passed as an `out` parameter, any previous value it refers to must be implicitly released. To give C++ mapping implementations enough hooks to meet this requirement, each object reference type results in the generation of an `out` type that is used solely as the `out` parameter type. For example, interface `TYPE` results in the object reference type `TYPE_ptr`, the helper type `TYPE_var`, and the `out` parameter type `TYPE_out`. The general form for object reference `out` types is as follows:

```c++
// C++
class TYPE_out
{

public:
    TYPE_out(TYPE_ptr& p) : ptr_(p) { ptr_ = TYPE::_nil(); }

    TYPE_out(TYPE_var & p) : ptr_(p.ptr_) {
```
release(ptr_); ptr_ = TYPE::_nil();
}

TYPE_out(TYPE_out& a) : ptr_(a.ptr_) {}

TYPE_out& operator=(TYPE_out& a) {
    ptr_ = a.ptr_; return *this;
}

TYPE_out& operator=(const TYPE_var& a) {
    ptr_ = TYPE::_duplicate(TYPE_ptr(a)); return *this;
}

operator TYPE_ptr&() { ptr_ = p; return *this; }

operator TYPE_ptr&() { return ptr_; }

private:
    TYPE_ptr& ptr_;
String outs

When a String_var is passed as an out parameter, any previous value it refers to must be implicitly freed. To give C++ mapping implementations enough hooks to meet this requirement, the string type also results in the generation of a String_out type in the CORBA namespace that is used solely as the string out parameter type. The general form for the String_out type is as follows:

```cpp
// C++
class String_out
{
  public:
    String_out(char*& p) : ptr_(p) { ptr_ = 0; }
    String_out(String_var& p) : ptr_(p.ptr_)
    {
      string_free(ptr_); ptr_ = 0;
    }
    String_out(String_out& s) : ptr_(s.ptr_) {}
    String_out& operator=(String_out& s) {
      ptr_ = s.ptr_; return *this;
    }
    String_out& operator=(char* p) {
      ptr_ = p; return *this;
    }
    String_out& operator=(const char* p) {
      ptr_ = string_dup(p); return *this;
    }
    operator char* &() { return ptr_; }
    char*& ptr() { return ptr_; }

  private:
    char*& ptr_;

    // assignment from String_var disallowed
    void operator=(const String_var &);
};
```

The first constructor binds the reference data member with the char* & argument. The second constructor binds the reference data member with the char* held by the String_var argument, and then calls string_free() on the string. The third constructor, the copy constructor, binds the reference data member to the same char* bound to the data member of its argument.
Assignment from another String_out copies the char* referenced by the argument String_out to the char* referenced by the data member. The overloaded assignment operator for char* simply assigns the char* argument to the data member. The overloaded assignment operator for const char* duplicates the argument and assigns the result to the data member.

Note that the assignment does not cause any previously held string to be freed; in this regard, the String_out type behaves exactly as a char*. The char*& conversion operator returns the data member. The ptr() member function, which can be used to avoid having to rely on implicit conversion, also returns the data member.

Assignment from String_var to a String_out is disallowed because of the memory management ambiguities involved. Specifically, it is not possible to determine whether the string owned by the String_var should be taken over by the String_out without copying, or if it should be copied. Disallowing assignment from String_var forces the application developer to make the choice explicitly, as follows:

```cpp
// C++
void
A::op(String_out arg)
{
    String_var s = string_dup("some string");
    ...
    out = s;               // disallowed; either
    out = string_dup(s);   // 1: copy, or
    out = s._retn();       // 2: adopt
}
```

On the line marked with the comment “1,” the caller is explicitly copying the string held by the String_var and assigning the result to the out argument. Alternatively, the caller could use the technique shown on the line marked with the comment “2” to force the String_var to give up its ownership of the string it holds so that it may be returned in the out argument without incurring memory management errors.

### Argument Passing Considerations

The mapping of parameter passing modes attempts to balance the need for both efficiency and simplicity. For primitive types, enumerations, and object references, the modes are straightforward, passing the type P for primitives and enumerations and the type A_ptr for an interface type A.
Aggregate types are complicated by the question of when and how parameter memory is allocated and deallocated. Mapping in parameters is straightforward because the parameter storage is caller-allocated and read-only. The mapping for out and inout parameters is more problematic. For variable-length types, the callee must allocate some if not all of the storage. For fixed-length types, such as a Point type represented as a struct containing three floating point members, caller allocation is preferable (to allow stack allocation).

To accommodate both kinds of allocation, avoid the potential confusion of split allocation, and eliminate confusion with respect to when copying occurs, the mapping is $T_\&$ for a fixed-length aggregate $T$ and $T_\&$ for a variable-length $T$. This approach has the unfortunate consequence that usage for structs depends on whether the struct is fixed- or variable-length; however, the mapping is consistently $T_\_var\&$ if the caller uses the managed type $T_\_var$.

The mapping for out and inout parameters additionally requires support for deallocating any previous variable-length data in the parameter when a $T_\_var$ is passed. Even though their initial values are not sent to the operation, the BEA Tuxedo includes out parameters because the parameter could contain the result from a previous call. The provision of the $T_\_out$ types is intended to give implementations the hooks necessary to free the inaccessible storage while converting from the $T_\_var$ types. The following examples demonstrate the compliant behavior:

```cpp
// IDL
struct S { string name; float age; };
void f(out S p);

// C++
S_\_var s;
f(s);
// use s
f(s);        // first result will be freed
S_\*sp;       // need not initialize before passing to out
f(sp);
// use sp
delete sp;   // cannot assume next call will free old value
f(sp);
```

Note that implicit deallocation of previous values for out and inout parameters works only with $T_\_var$ types, not with other types:
// IDL
void q(out string s);

// C++
char *s;
for (int i = 0; i < 10; i++)
q(s);        // memory leak!

Each call to the q function in the loop results in a memory leak because the caller is not invoking
string_free on the out result. There are two ways to fix this, as shown below:

// C++
char *s;
String_var svar;
for (int i = 0 ; i < 10; i++) {
    q(s);
    string_free(s);  // explicit deallocation
    // OR:
    q(svar);        // implicit deallocation
}

Using a plain char* for the out parameter means that the caller must explicitly deallocate its
memory before each reuse of the variable as an out parameter, while using a String_var means
that any deallocation is performed implicitly upon each use of the variable as an out parameter.

Variable-length data must be explicitly released before being overwritten. For example, before
assigning to an inout string parameter, the implementor of an operation may first delete the old
character data. Similarly, an inout interface parameter should be released before being
reassigned. One way to ensure that the parameter storage is released is to assign it to a local T_var
variable with an automatic release, as in the following example:

// IDL
interface A;
void f(inout string s, inout A obj);

// C++
void A_impl::f(char *&s, A_ptr &obj) {
    String_var s_tmp = s;
    s = /* new data */;
    A_var obj_tmp = obj;
    obj = /* new reference */
}
For parameters that are passed or returned as a pointer (T*) or as a reference to a pointer (T* &), an application is not allowed to pass or return a NULL pointer; the result of doing so is undefined. In particular, a caller may not pass a NULL pointer under any of the following circumstances:

- in and inout string
- in and inout array (pointer to first element)

However, a caller may pass a reference to a pointer with a NULL value for out parameters, because the callee does not examine the value, but overwrites it. A callee may not return a NULL pointer under any of the following circumstances:

- out and return variable-length struct
- out and return variable-length union
- out and return string
- out and return sequence
- out and return variable-length array, return fixed-length array
- out and return any

### Operation Parameters and Signatures

Table 13-7 displays the mapping for the basic OMG IDL parameter passing modes and return type according to the type being passed or returned. Table 13-8 displays the same information for T_var types. Table 13-8 is merely for informational purposes; it is expected that operation signatures for both clients and servers will be written in terms of the parameter-passing modes shown in Table 13-7, with the exception that the T_out types will be used as the actual parameter types for all out parameters.

It is also expected that T_var types will support the necessary conversion operators to allow them to be passed directly. Callers should always pass instances of either T_var types or the base types shown in Table 13-7, and callees should treat their T_out parameters as if they were actually the corresponding underlying types shown in Table 13-7.

In Table 13-7, fixed-length arrays are the only case where the type of an out parameter differs from a return value, which is necessary because C++ does not allow a function to return an array. The mapping returns a pointer to a slice of the array, where a slice is an array with all the dimensions of the original array specified except the first dimension.
<table>
<thead>
<tr>
<th>Data Type</th>
<th>In</th>
<th>Inout</th>
<th>Out</th>
<th>Return</th>
</tr>
</thead>
<tbody>
<tr>
<td>short</td>
<td>Short</td>
<td>Short&amp;</td>
<td>Short&amp;</td>
<td>Short</td>
</tr>
<tr>
<td>long</td>
<td>Long</td>
<td>Long&amp;</td>
<td>Long&amp;</td>
<td>Long</td>
</tr>
<tr>
<td>unsigned short</td>
<td>UShort</td>
<td>UShort&amp;</td>
<td>UShort&amp;</td>
<td>UShort</td>
</tr>
<tr>
<td>unsigned long</td>
<td>ULong</td>
<td>ULong&amp;</td>
<td>ULong&amp;</td>
<td>ULong</td>
</tr>
<tr>
<td>float</td>
<td>Float</td>
<td>Float&amp;</td>
<td>Float&amp;</td>
<td>Float</td>
</tr>
<tr>
<td>double</td>
<td>Double</td>
<td>Double&amp;</td>
<td>Double&amp;</td>
<td>Double</td>
</tr>
<tr>
<td>boolean</td>
<td>Boolean</td>
<td>Boolean&amp;</td>
<td>Boolean&amp;</td>
<td>Boolean</td>
</tr>
<tr>
<td>char</td>
<td>Char</td>
<td>Char&amp;</td>
<td>Char&amp;</td>
<td>Char</td>
</tr>
<tr>
<td>wchar</td>
<td>WChar</td>
<td>WChar&amp;</td>
<td>WChar</td>
<td>Octet</td>
</tr>
<tr>
<td>octet</td>
<td>Octet</td>
<td>Octet&amp;</td>
<td>Octet&amp;</td>
<td>Octet</td>
</tr>
<tr>
<td>enum</td>
<td>enum</td>
<td>enum&amp;</td>
<td>enum&amp;</td>
<td>enum</td>
</tr>
<tr>
<td>object reference ptr</td>
<td>objref_ptr</td>
<td>objref_ptr&amp;</td>
<td>objref_ptr&amp;</td>
<td>objref_ptr</td>
</tr>
<tr>
<td>struct, fixed</td>
<td>const struct&amp;</td>
<td>struct&amp;</td>
<td>struct&amp;</td>
<td>struct</td>
</tr>
<tr>
<td>struct, variable</td>
<td>const struct&amp;</td>
<td>struct&amp;</td>
<td>struct* &amp;</td>
<td>struct*</td>
</tr>
<tr>
<td>union, fixed</td>
<td>const union&amp;</td>
<td>union&amp;</td>
<td>union&amp;</td>
<td>union</td>
</tr>
<tr>
<td>union, variable</td>
<td>const union&amp;</td>
<td>union&amp;</td>
<td>union* &amp;</td>
<td>union*</td>
</tr>
<tr>
<td>string</td>
<td>const char*</td>
<td>char* &amp;</td>
<td>char* &amp;</td>
<td>char*</td>
</tr>
<tr>
<td>wstring</td>
<td>const WChar</td>
<td>WChar* &amp;</td>
<td>WChar* &amp;</td>
<td>WChar*</td>
</tr>
<tr>
<td>sequence</td>
<td>const sequence&amp;</td>
<td>sequence&amp;</td>
<td>sequence* &amp;</td>
<td>sequence*</td>
</tr>
<tr>
<td>array, fixed</td>
<td>const array</td>
<td>array</td>
<td>array</td>
<td>array slice* (See Note below.)</td>
</tr>
</tbody>
</table>
Note: The Object reference ptr data type includes pseudo-object references. The array slice return is an array with all the dimensions of the original array except the first dimension. A caller is responsible for providing storage for all arguments passed as in arguments.

Table 13-7 Basic Argument and Result Passing (Continued)

<table>
<thead>
<tr>
<th>Data Type</th>
<th>In</th>
<th>Inout</th>
<th>Out</th>
<th>Return</th>
</tr>
</thead>
<tbody>
<tr>
<td>array, variable</td>
<td>const array</td>
<td>array</td>
<td>array slice* &amp;</td>
<td>array slice*</td>
</tr>
<tr>
<td>any</td>
<td>const any &amp;</td>
<td>any &amp;</td>
<td>any &amp;</td>
<td>any &amp;</td>
</tr>
</tbody>
</table>

Table 13-8 T_var Argument and Result Passing

<table>
<thead>
<tr>
<th>Data Type</th>
<th>In</th>
<th>Inout</th>
<th>Out</th>
<th>Return</th>
</tr>
</thead>
<tbody>
<tr>
<td>object reference var</td>
<td>const objref_var &amp;</td>
<td>objref_var &amp;</td>
<td>objref_var &amp;</td>
<td>objref_var</td>
</tr>
<tr>
<td>struct_var</td>
<td>const struct_var &amp;</td>
<td>struct_var &amp;</td>
<td>struct_var &amp;</td>
<td>struct_var</td>
</tr>
<tr>
<td>union_var</td>
<td>const union_var &amp;</td>
<td>union_var &amp;</td>
<td>union_var &amp;</td>
<td>union_var</td>
</tr>
<tr>
<td>string_var</td>
<td>const string_var &amp;</td>
<td>string_var &amp;</td>
<td>string_var &amp;</td>
<td>string_var</td>
</tr>
<tr>
<td>sequence_var</td>
<td>const sequence_var &amp;</td>
<td>sequence_var &amp;</td>
<td>sequence_var &amp;</td>
<td>sequence_var</td>
</tr>
<tr>
<td>array_var</td>
<td>const array_var &amp;</td>
<td>array_var &amp;</td>
<td>array_var &amp;</td>
<td>array_var</td>
</tr>
<tr>
<td>any_var</td>
<td>const any_var &amp;</td>
<td>any_var &amp;</td>
<td>any_var &amp;</td>
<td>any_var</td>
</tr>
</tbody>
</table>

Note: The object reference var data type includes pseudo-object references. Table 13-9 and Table 13-10 describe the caller’s responsibility for storage associated with inout and out parameters and for return results.

Table 13-9 Caller Argument Storage Responsibilities

<table>
<thead>
<tr>
<th>Type</th>
<th>Inout Param</th>
<th>Out Param</th>
<th>Return Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>short</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>long</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Type</td>
<td>Inout Param</td>
<td>Out Param</td>
<td>Return Result</td>
</tr>
<tr>
<td>-----------------------</td>
<td>-------------</td>
<td>-----------</td>
<td>---------------</td>
</tr>
<tr>
<td>unsigned short</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>unsigned long</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>float</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>double</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>boolean</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>char</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>wchar</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>octet</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>enum</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>object reference ptr</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>struct, fixed</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>struct, variable</td>
<td>1</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>union, fixed</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>union, variable</td>
<td>1</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>string</td>
<td>4</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>wstring</td>
<td>4</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>sequence</td>
<td>5</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>array, fixed</td>
<td>1</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>array, variable</td>
<td>1</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>any</td>
<td>5</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>
## Table 13-10 Argument Passing Cases

<table>
<thead>
<tr>
<th>Case</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Caller allocates all necessary storage, except that which may be encapsulated and managed within the parameter itself. For inout parameters, the caller provides the initial value, and the callee may change that value. For out parameters, the caller allocates the storage but need not initialize it, and the callee sets the value. Function returns are by value.</td>
</tr>
<tr>
<td>2</td>
<td>Caller allocates storage for the object reference. For inout parameters, the caller provides an initial value; if the callee wants to reassign the inout parameter, it will first call CORBA::release on the original input value. To continue to use an object reference passed in as an inout, the caller must first duplicate the reference. The caller is responsible for the release of all out and return object references. Release of all object references embedded in other structures is performed automatically by the structures themselves.</td>
</tr>
<tr>
<td>3</td>
<td>For out parameters, the caller allocates a pointer and passes it by reference to the callee. The callee sets the pointer to point to a valid instance of the parameter’s type. For returns, the callee returns a similar pointer. The callee is not allowed to return a NULL pointer in either case. In both cases, the caller is responsible for releasing the returned storage. To maintain local/remote transparency, the caller must always release the returned storage, regardless of whether the callee is located in the same address space as the caller or is located in a different address space. Following the completion of a request, the caller is not allowed to modify any values in the returned storage—to do so, the caller must first copy the returned instance into a new instance, and modify the new instance.</td>
</tr>
<tr>
<td>4</td>
<td>For inout strings, the caller provides storage for both the input string and the char* pointing to it. Since the callee may deallocate the input string and reassign the char* to point to new storage to hold the output value, the caller should allocate the input string using string_alloc(). The size of the out string is, therefore, not limited by the size of the in string. The caller is responsible for deleting the storage for the out using string_free(). The callee is not allowed to return a NULL pointer for an inout, out, or return value.</td>
</tr>
</tbody>
</table>
5  For inout sequences and anys, assignment or modification of the sequence or any may cause deallocation of owned storage before any reallocation occurs, depending upon the state of the Boolean release parameter with which the sequence or any was constructed.

6  For out parameters, the caller allocates a pointer to an array slice, which has all the same dimensions of the original array except the first, and passes the pointer by reference to the callee. The callee sets the pointer to point to a valid instance of the array.

For returns, the callee returns a similar pointer. The callee is not allowed to return a NULL pointer in either case. In both cases, the caller is responsible for releasing the returned storage.

To maintain local/remote transparency, the caller must always release the returned storage, regardless of whether the callee is located in the same address space as the callee or is located in a different address space. Following completion of a request, the caller is not allowed to modify any values in the returned storage—to do so, the caller must first copy the returned array instance into a new array instance, and modify the new instance.
This chapter describes the BEA Tuxedo implementation of the CORBA core member functions in C++ and their extensions. It also describes pseudo-objects and their relationship to C++ classes. Pseudo-objects are object references that cannot be transmitted across the network. Pseudo-objects are similar to other objects; however, because the ORB owns them, they cannot be extended.

Notes: Some of the information in this chapter is taken from the Common Object Request Broker: Architecture and Specification. Revision 2.4.2, February 2001, published by the Object Management Group (OMG). Used with permission of the OMG.

The BEA Tuxedo CORBA Java client and BEA Tuxedo CORBA Java client ORB were deprecated in Tuxedo 8.1 and are no longer supported in Tuxedo 9.x. All BEA Tuxedo CORBA Java client and BEA Tuxedo CORBA Java client ORB text references, associated code samples, etc. should only be used:

- to help implement/run third party Java ORB libraries, and
- for programmer reference only.

Technical support for third party CORBA Java ORBs should be provided by their respective vendors. BEA Tuxedo does not provide any technical support or documentation for third party CORBA Java ORBs.

**Global Classes**

The following BEA Tuxedo classes are global in scope:

- CORBA
• Tobj

These classes contain the predefined types, classes, and functions used in BEA Tuxedo development.

The CORBA class contains the classes, data types, and member functions essential to using an Object Request Broker (ORB) as defined by CORBA. The BEA Tuxedo extensions to CORBA are contained in the Tobj C++ class. The Tobj class contains data types, nested classes, and member functions that BEA Tuxedo provides as an extension to CORBA.

Using CORBA data types and member functions in the BEA Tuxedo product requires the CORBA:: prefix. For example, a Long is a CORBA::Long. Likewise, to use Tobj nested classes and member functions in the BEA Tuxedo product, you need the Tobj:: prefix. For example, FactoryFinder is Tobj::FactoryFinder.

Pseudo-objects

Pseudo-objects are represented as local classes, which reside in the CORBA class. A pseudo-object and its corresponding member functions are named using a nested class structure. For example, an ORB object is a CORBA::ORB and a Current object is a CORBA::Current.

Any Class Member Functions

This section describes the member functions of the Any class.

The mapping of these member functions to C++ is as follows:

```cpp
class CORBA
{
    class Any
    {
    public:
        Any ();
        Any (const Any&);
        Any (TypeCode_ptr tc, void *value, Boolean release =
             CORBA_ FALSE);
        ~Any ();
        Any & operator=(const Any&);

        void    operator<<(Short);
        void    operator<<(UShort);
```

```cpp```
Any Class Member Functions

void    operator<<=(Long);
void    operator<<=(ULong);
void    operator<<=(Float);
void    operator<<=(Double);
void    operator<<=(const Any&);
void    operator<<=(const char*);
void    operator<<=(Object_ptr);
void    operator<<=(from_boolean);
void    operator<<=(from_char);
void    operator<<=(from_octet);
void    operator<<=(from_string);
Boolean  operator>>=(Short&) const;
Boolean  operator>>=(UShort&) const;
Boolean  operator>>=(Long&) const;
Boolean  operator>>=(ULong&) const;
Boolean  operator>>=(Float&) const;
Boolean  operator>>=(Double&) const;
Boolean  operator>>=(Any&) const;
Boolean  operator>>=(char*&) const;
Boolean  operator>>=(Object_ptr&) const;
Boolean  operator>>=(to_boolean) const;
Boolean  operator>>=(to_char) const;
Boolean  operator>>=(to_octet) const;
Boolean  operator>>=(to_object) const;
Boolean  operator>>=(to_string) const;

TypeCode_ptr type() const;
void        replace(TypeCode_ptr, void *, Boolean);
void        replace(TypeCode_ptr, void *);
const void * value() const;

}; //CORBA

CORBA::Any::Any()

Synopsis
Constructs the Any object.
C++ Binding

    CORBA::Any::Any()

Arguments

None.

Description

This is the default constructor for the CORBA::Any class. It creates an Any object with a TypeCode of type tc_null and a value of 0 (zero).

Return Values

None.

CORBA::Any::Any(const CORBA::Any & InitAny)

Synopsis

Constructs the Any object that is a copy of another Any object.

C++ Binding

    CORBA::Any::Any(const CORBA::Any & InitAny)

Argument

    InitAny
     Refers to the CORBA::Any to copy.

Description

This is the copy constructor for the CORBA::Any class. This constructor duplicates the TypeCode reference of the Any that is passed in.

The type of copying to be performed is determined by the release flag of the Any object to be copied. If release evaluates as CORBA_TRUE, the constructor deep-copies the parameter’s value; if release evaluates as CORBA_FALSE, the constructor shallow-copies the parameter’s value. Using a shallow copy gives you more control to optimize memory allocation, but the caller must ensure the Any does not use memory that has been freed.
Any Class Member Functions

Return Values
None.

CORBA::Any::Any(TypeCode_ptr TC, void * Value, Boolean Release)

Synopsis
Creates the Any object using a TypeCode and a value.

C++ Binding
CORBA::Any::Any(TypeCode_ptr TC, void * Value, Boolean Release)

Arguments
TC
A pointer to a TypeCode pseudo-object reference, specifying the type to be created.

Value
A pointer to the data to be used to create the Any object. The data type of this argument must match the TypeCode specified.

Release
Determines whether the Any assumes ownership of the memory specified by the Value argument. If Release is CORBA_TRUE, the Any assumes ownership. If Release is CORBA_FALSE, the Any does not assume ownership; the data pointed to by the Value argument is not released upon assignment or destruction.

Description
This constructor is used with the nontype-safe Any interface. It duplicates the specified TypeCode object reference and then inserts the data pointed to by value inside the Any object.

Return Values
None.

CORBA::Any::~Any()

Synopsis
Destructor for the Any.
C++ Binding

CORBA::Any::~Any()

Arguments
None.

Description
This destructor frees the memory that the CORBA::Any holds (if the Release flag is specified as CORBA_TRUE), and releases the TypeCode pseudo-object reference contained in the Any.

Return Values
None.

CORBA::Any & CORBA::Any::operator=(const CORBA::Any & InitAny)

Synopsis
Any assignment operator.

C++ Binding

CORBA::Any & CORBA::Any::operator=(const CORBA::Any & InitAny)

Arguments
InitAny
A reference to an Any to use in the assignment. The Any to use in the assignment determines whether the Any assumes ownership of the memory in Value. If Release is CORBA_TRUE, the Any assumes ownership and deep-copies the InitAny argument’s value; if Release is CORBA_FALSE, the Any shallow-copies the InitAny argument’s value.

Description
This is the assignment operator for the Any class. Memory management of this member function is determined by the current value of the Release flag. The current value of the Release flag determines whether the current memory is released before the assignment. If the current Release flag is CORBA_TRUE, the Any releases any value previously held; if the current Release flag is CORBA_FALSE, the Any does not release any value previously held.
Any Class Member Functions

Return Values

Returns the Any, which holds the copy of the InitAny.

void CORBA::any::operator<==()

Synopsis

Type safe Any insertion operators.

C++ Binding

void CORBA::Any::operator<==(CORBA::Short Value)
void CORBA::Any::operator<==(CORBA::UShort Value)
void CORBA::Any::operator<==(CORBA::Long Value)
void CORBA::Any::operator<==(CORBA::Ulong Value)
void CORBA::Any::operator<==(CORBA::Float Value)
void CORBA::Any::operator<==(CORBA::Double Value)
void CORBA::Any::operator<==(const CORBA::Any & Value)
void CORBA::Any::operator<==(const char * Value)
void CORBA::Any::operator<==(Object_ptr Value)

Argument

Value

Type specific value to be inserted into the Any.

Description

This insertion member function performs type-safe insertions. If the Any had a previous value, and the Release flag is CORBA_TRUE, the memory is deallocated and the previous TypeCode object reference is freed. The new value is inserted into the Any by copying the value passed in using the Value parameter. The appropriate TypeCode reference is duplicated.

Return Values

None.

CORBA::Boolean CORBA::Any::operator>=()

Synopsis

Type safe Any extraction operators.
C++ Binding

```cpp
CORBA::Boolean CORBA::Any::operator>>(
    CORBA::Short & Value) const
CORBA::Boolean CORBA::Any::operator>>(
    CORBA::UShort & Value) const
CORBA::Boolean CORBA::Any::operator>>(
    CORBA::Long & Value) const
CORBA::Boolean CORBA::Any::operator>>(
    CORBA::Ulong & Value) const
CORBA::Boolean CORBA::Any::operator>>(
    CORBA::Float & Value) const
CORBA::Boolean CORBA::Any::operator>>(
    CORBA::Double & Value) const
CORBA::Boolean CORBA::Any::operator>>(
    CORBA::Any & Value) const
CORBA::Boolean CORBA::Any::operator>>(
    char * & Value) const
CORBA::Boolean CORBA::Any::operator>>(
    Object_ptr & Value) const
```

Argument

The `Value` argument is a reference to the relevant object that receives the output of the value contained in the `Any` object.

Description

This extraction member function performs type-safe extractions. If the `Any` object contains the specified type, this member function assigns the pointer of the `Any` to the output reference value, `Value`, and `CORBA_TRUE` is returned. If the `Any` does not contain the appropriate type, `CORBA_FALSE` is returned. The caller must not attempt to release or delete the storage because it is owned and managed by the `Any` object. The `Value` argument is a reference to the relevant object that receives the output of the value contained in the `Any` object. If the `Any` object does not contain the appropriate type, the value remains unchanged.

Return Values

`CORBA_TRUE` if the `Any` contained a value of the specific type. `CORBA_FALSE` if the `Any` did not contain a value of the specific type.
**CORBA::Any::operator<==()**

**Synopsis**
Type safe insertion operators for Any.

**C++ Binding**
void CORBA::Any::operator<== (from_boolean Value)  
void CORBA::Any::operator<== (from_char Value)  
void CORBA::Any::operator<== (from_octet Value)  
void CORBA::Any::operator<== (from_string Value)

**Argument**
Value  
A relevant object that contains the value to insert into the Any.

**Description**
These insertion member functions perform a type-safe insertion of a CORBA::Boolean, a CORBA::Char, or a CORBA::Octet reference into an Any. If the Any had a previous value, and its Release flag is CORBA_TRUE, the memory is deallocated and the previous TypeCode object reference is freed. The new value is inserted into the Any object by copying the value passed in using the Value parameter. The appropriate TypeCode reference is duplicated.

**Return Values**
None.

**CORBA::Boolean CORBA::Any::operator>>=()**

**Synopsis**
Type-safe extraction operators for Any.

**C++ Binding**
CORBA::Boolean CORBA::Any::operator >>= (to_boolean Value) const  
CORBA::Boolean CORBA::Any::operator >>= (to_char Value) const  
CORBA::Boolean CORBA::Any::operator >>= (to_octet Value) const  
CORBA::Boolean CORBA::Any::operator >>= (to_object Value) const  
CORBA::Boolean CORBA::Any::operator >>= (to_string Value) const
Argument

Value
A reference to the relevant object that receives the output of the value contained in the Any object. If the Any object does not contain the appropriate type, the value remains unchanged.

Description
These extraction member functions perform a type-safe extraction of a CORBA::Boolean, a CORBA::Char, a CORBA::Octet, a CORBA::Object, or a String reference from an Any. These member functions are helpers nested in the Any class. Their purpose is to distinguish extractions of the OMG IDL types: Boolean, char, and octet (C++ does not require these to be distinct types).

Return Values
If the Any contains the specified type, this member function assigns the value in the Any object reference to the output variable, Value, and returns CORBA_TRUE. If the Any object does not contain the appropriate type, CORBA_FALSE is returned.

CORBA::TypeCode_ptr CORBA::Any::type() const

Synopsis
TypeCode accessor for Any.

C++ Binding
CORBA::TypeCode_ptr CORBA::Any::type();

Arguments
None.

Description
This function returns the TypeCode_ptr pseudo-object reference of the TypeCode object associated with the Any. The TypeCode_ptr pseudo-object reference must be released by the CORBA::release member function or must be assigned to a TypeCode_var to be automatically released.

Return Values
TypeCode_ptr contained in the Any.
void CORBA::Any::replace()

Synopsis
Nontype safe Any “insertion.”

C++ Binding
void CORBA::Any::replace(TypeCode_ptr TC, void * Value,
       Boolean Release = CORBA_FALSE);

Arguments
TC
A TypeCode pseudo-object reference specifying the TypeCode value for the replaced Any object. This argument is duplicated.

Value
A void pointer specifying the storage pointed to by the Any object.

Release
Determines whether the Any manages the specified Value argument. If Release is CORBA_TRUE, the Any assumes ownership. If Release is CORBA_FALSE, the Any does not assume ownership and the data pointed to by the Value parameter is not released upon assignment or destruction.

Description
These member functions replace the data and TypeCode value currently contained in the Any with the value of the TC and Value arguments passed in. The functions perform a nontype-safe replacement, which means that the caller is responsible for consistency between the TypeCode value and the data type of the storage pointed to by the Value argument.

If the value of Release is CORBA_TRUE, this function releases the existing TypeCode pseudo-object in the Any object and frees the storage pointed to be the Any object reference.

Return Values
None.

Context Member Functions
A Context supplies optional context information associated with a method invocation.

The mapping of these member functions to C++ is as follows:
Memory Management

Context has the following special memory management rule:

- Ownership of the return values of the `context_name` and `parent` functions is maintained by the Context; these return values must not be freed by the caller.

This section describes Context member functions.

**CORBA::Context::context_name**

**Synopsis**

Returns the name of a given Context object.

**C++ Binding**

```cpp
const char * CORBA::Context::context_name() const;
```
Arguments

None.

Description

This member function returns the name of a given Context object. The Context object reference owns the memory for the returned `char *`. Users should not modify this memory.

Return Values

If the member function succeeds, it returns the name of the Context object. The value may be empty if the Context object is not a child Context created by a call to `CORBA::Context::create_child`.

If the Context object has no name, this is an empty string.

**CORBA::Context::create_child**

Synopsis

Creates a child of the Context object.

C++ Binding

```cpp
void CORBA::Context::create_child ( 
    const char *              CtxName, 
    CORBA::Context_out        CtxObject);
```

Arguments

- **CtxName**
  
  The name to be associated with the child of the Context reference.

- **CtxObject**
  
  The newly created Context object reference.

Exception

- `CORBA::NO_MEMORY`

Description

This member function creates a child of the Context object. That is, searches on the child Context object will look for matching property names in the parent context (and so on, up the context tree), if necessary.
Return Values
None.

See Also
CORBA::ORB::get_default_context
CORBA::release

**CORBA::Context::delete_values**

Synopsis
Deletes the values for a specified attribute in the Context object.

C++ Binding

```c++
void CORBA::Context::delete_values (const char * AttrName);
```

Argument

- **AttrName**
  The name of the attribute whose values are to be deleted. If this argument has a trailing wildcard character (*), all names that match the string preceding the wildcard character are deleted.

Exceptions

- **CORBA::BAD_PARAM** if attribute is an empty string.
- **CORBA::BAD_CONTEXT** if no matching attributes to be deleted were found.

Description
This member function deletes named values for an attribute in the Context object. Note that it does not recursively do the same to its parents, if any.

Return Values
None.

See Also
CORBA::Context::create_child
CORBA::ORB::get_default_context
**CORBA::Context::get_values**

**Synopsis**
Retrieves the values for a given attribute in the Context object within the specified scope.

**C++ Binding**
```cpp
void CORBA::Context::get_values (  
    const char * StartScope,  
    CORBA::Flags OpFlags,  
    const char * AttrName,  
    CORBA::NVList_out AttrValues);
```

**Arguments**
- **StartScope**
  The Context object level at which to initiate the search for specified properties. The level is the name of the context, or `parent`, at which the search is started. If the value is 0 (zero), the search begins with the current Context object.

- **OpFlags**
  The only valid operation flag is `CORBA::CTX_RESTRICT_SCOPE`. If you specify this flag, the object implementation restricts the property search to the current scope only (that is, the property search is not executed recursively up the chain of the parent context); otherwise, the search continues to a wider scope until a match has been found or until all wider levels have been searched.

- **AttrName**
  The name of the attribute whose values are to be returned. If this argument has a trailing wildcard character (*), all names that match the string preceding the wildcard character are returned.

- **AttrValues**
  Receives the values for the specified attributes (returns an `NVList` object) where each item in the list is a `NamedValue`.

**Exceptions**
- `CORBA::BAD_PARAM` if attribute is an empty string.
- `CORBA::BAD_CONTEXT` if no matching attributes were found.
- `CORBA::NO_MEMORY` if dynamic memory allocation failed.
Description

This member function retrieves the values for a specified attribute in the Context object. These values are returned as an NVList object, which must be freed when no longer needed using the CORBA::release member function.

Return Values

None.

See Also

CORBA::Context::create_child
CORBA::ORB::get_default_context

CORBA::Context::parent

Synopsis

Returns the parent context of the Context object.

C++ Binding

CORBA::Context_ptr CORBA::Context::parent () const;

Arguments

None.

Description

This member function returns the parent context of the Context object. The parent of the Context object is an attribute owned by the Context and should not be modified or freed by the caller. This parent is nil unless the Context object was created using the CORBA::Context::create_child member function.

Return Values

If the member function succeeds, the parent context of the Context object is returned. The parent context may be nil. Use the CORBA::is_nil member function to test for a nil object reference.

If the member function does not succeed, an exception is thrown. Use the CORBA::is_nil member function to test for a nil object reference.
**CORBA::Context::set_one_value**

**Synopsis**
Sets the value for a given attribute in the Context object.

**C++ Binding**
```cpp
void CORBA::Context::set_one_value (
    const char *            AttrName,
    const CORBA::Any &       AttrValue);
```

**Arguments**
- **AttrName**
  The name of the attribute to set.
- **AttrValue**
  The value of the attribute. Currently, the BEA Tuxedo system supports only the string type; therefore, this parameter must contain a `CORBA::Any` object with a string inside.

**Exceptions**
- `CORBA::BAD_PARAM` if AttrName is an empty string or AttrValue does not contain a string type.
- `CORBA::NO_MEMORY` if dynamic memory allocation failed.

**Description**
This member function sets the value for a given attribute in the Context object. Currently, only string values are supported by the Context object. If the Context object already has an attribute with the given name, it is deleted first.

**Return Values**
None.

**See Also**
- `CORBA::Context::get_values`
- `CORBA::Context::set_values`
**CORBA::Context::set_values**

**Synopsis**
Sets the values for given attributes in the Context object.

**C++ Binding**
```cpp
void CORBA::Context::set_values (
    CORBA::NVList_ptr AttrValue);
```

**Argument**

*AttrValues*
The name and value of the attribute. Currently the BEA Tuxedo system supports only the string type; therefore, all NamedValue objects in the list must have CORBA::Any objects with a string inside.

**Exceptions**
- CORBA::BAD_PARAM if any of the attribute values has a value that is not a string type.
- CORBA::NO_MEMORY if dynamic memory allocation failed.

**Description**
This member function sets the values for given attributes in the Context object. The CORBA::NVList member function contains the property name and value pairs to be set.

**Return Values**
None.

**See Also**
- CORBA::Context::get_values
- CORBA::Context::set_one_value

---

**ContextList Member Functions**

The ContextList allows a client or server application to provide a list of context strings that must be supplied with Request invocation. For a description of the Request member functions, see the section “Request Member Functions” on page 14-95.

The ContextList differs from the Context in that the former supplies only the context strings whose values are to be looked up and sent with the request invocation (if applicable), while the
latter is where those values are obtained. For a description of the Context member functions, see the section Context Member Functions.

The mapping of these member functions to C++ is as follows:

```cpp
class CORBA
{
    class ContextList
    {
        public:
            Ulong count ();
            void add(const char* ctxt);
            void add_consume(char* ctxt);
            const char* item(Ulong index);
            Status remove(Ulong index);
    }; // ContextList
}; // CORBA
```

**CORBA::ContextList:: count**

**Synopsis**

Retrieves the current number of items in the list.

**C++ Binding**

```cpp
Ulong count ();
```

**Arguments**

None.

**Exception**

If the function does not succeed, an exception is thrown.

**Description**

This member function retrieves the current number of items in the list.
Return Values

If the function succeeds, the returned value is the number of items in the list. If the list has just been created, and no ContextList objects have been added, this function returns 0 (zero).

See Also

CORBA::ContextList::add
CORBA::ContextList::add_consume
CORBA::ContextList::item
CORBA::ContextList::remove

CORBA::ContextList::add

Synopsis

Constructs a ContextList object with an unnamed item, setting only the flags attribute.

C++ Binding

void add(const char* ctxt);

Argument

ctxt
    Defines the memory location referred to by char*.

Exception

If the member function does not succeed, a CORBA::NO_MEMORY exception is thrown.

Description

This member function constructs a ContextList object with an unnamed item, setting only the flags attribute.

The ContextList object grows dynamically; your application does not need to track its size.

Return Values

If the function succeeds, the return value is a pointer to the newly created ContextList object.

See Also

CORBA::ContextList::add_consume
CORBA::ContextList::count
**ContextList Member Functions**

**CORBA::ContextList::item**

**Synopsis**

Retrieves a pointer to the ContextList object, based on the index passed in.

**CORBA::ContextList::add_consume**

**Synopsis**

Constructs a ContextList object.

**C++ Binding**

```cpp
void add_consume(const char* ctxt);
```

**Argument**

- **ctxt**
  
  Defines the memory location referred to by char*.

**Exception**

If the member function does not succeed, an exception is raised.

**Description**

This member function constructs a ContextList object.

The ContextList object grows dynamically; your application does not need to track its size.

**Return Values**

If the function succeeds, the return value is a pointer to the newly created ContextList object.

**See Also**

- CORBA::ContextList::add
- CORBA::ContextList::count
- CORBA::ContextList::item
- CORBA::ContextList::remove

**CORBA::ContextList::remove**
**C++ Binding**

```cpp
const char* item(ULong index);
```

**Argument**

- **index**
  
  The index into the ContextList object. The indexing is zero-based.

**Exceptions**

- If this function does not succeed, the **BAD_PARAM** exception is thrown.

**Description**

This member function retrieves a pointer to a ContextList object, based on the index passed in. The function uses zero-based indexing.

**Return Values**

- If the function succeeds, the return value is a pointer to the ContextList object.

**See Also**

- CORBA::ContextList::add
- CORBA::ContextList::add_consume
- CORBA::ContextList::count
- CORBA::ContextList::remove

**CORBA::ContextList::remove**

**Synopsis**

Removes the item at the specified index, frees any associated memory, and reorders the remaining items on the list.

**C++ Binding**

```cpp
Status remove(ULong index);
```

**Argument**

- **Index**
  
  The index into the ContextList object. The indexing is zero-based.
Exceptions

If this function does not succeed, the BAD_PARAM exception is thrown.

Description

This member function removes the item at the specified index, frees any associated memory, and reorders the remaining items on the list.

Return Values

None.

See Also

CORBA::ContextList::add
CORBA::ContextList::add_consume
CORBA::ContextList::count
CORBA::ContextList::item

NamedValue Member Functions

NamedValue is used only as an element of NVList, especially in the DII. NamedValue maintains an (optional) name, an any value, and labelling flags. Legal flag values are CORBA::ARG_IN, CORBA::ARG_OUT, and CORBA::ARG_INOUT.

The value in a NamedValue may be manipulated via standard operations on any.

The mapping of these member functions to C++ is as follows:

```cpp
// C++
class NamedValue
{
    public:
        Flags        flags() const;
        const char * name() const;
        Any *        value() const;
};
```

Memory Management

NamedValue has the following special memory management rule:
Ownership of the return values of the `name()` and `value()` functions is maintained by the NamedValue; these return values must not be freed by the caller.

The following sections describe NamedValue member functions.

**CORBA::NamedValue::flags**

Synopsis

Retrieves the flags attribute of the NamedValue object.

C++ Binding

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

Arguments

None.

Description

This member function retrieves the flags attribute of the NamedValue object.

Return Values

If the function succeeds, the return value is the flags attribute of the NamedValue object.

If the function does not succeed, an exception is thrown.

**CORBA::NamedValue::name**

Synopsis

Retrieves the name attribute of the NamedValue object.

C++ Binding

```cpp
const char * CORBA::NamedValue::name () const;
```
NamedValue Member Functions

Arguments

None.

Description

This member function retrieves the name attribute of the NamedValue object. The name returned by this member function is owned by the NamedValue object and should not be modified or released.

Return Values

If the function succeeds, the value returned is a constant Identifier object representing the name attribute of the NamedValue object.

If the function does not succeed, an exception is thrown.

**CORBA::NamedValue::value**

Synopsis

Retrieves a pointer to the value attribute of the NamedValue object.

C++ Binding

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

Arguments

None.

Description

This member function retrieves a pointer to the `Any` object that represents the value attribute of the NamedValue object. This attribute is owned by the NamedValue object, and should not be modified or released.

Return Values

If the function succeeds, the return value is a pointer to the `Any` object contained in the NamedValue object.

If the function does not succeed, an exception is thrown.
NVList Member Functions

NVList is a list of NamedValues. A new NVList is constructed using the ORB::create_list operation (see CORBA::ORB::create_exception_list). New NamedValues may be constructed as part of an NVList, in any of the following ways:

- `add`—creates an unnamed value, initializing only the flags
- `add_item`—initializes name and flags
- `add_value`—initializes name, value, and flags

Each of these operations returns the new item.

Elements may be accessed and deleted via zero-based indexing. The `add`, `add_item`, `add_value`, `add_item_consume`, and `add_value_consume` functions lengthen the NVList to hold the new element each time they are called. The `item` function can be used to access existing elements.

```cpp
// C++
class NVList
{
  public:
    ULong count() const;
    NamedValue_ptr add(Flags);
    NamedValue_ptr add_item(const char*, Flags);
    NamedValue_ptr add_value(const char*, const Any&, Flags);
    NamedValue_ptr item(ULong);
    void remove(ULong);
};
```

Memory Management

NVList has the following special memory management rules:

- Ownership of the return values of the `add`, `add_item`, `add_value`, `add_item_consume`, `add_value_consume`, and `item` functions is maintained by the NVList; these return values must not be freed by the caller.

- The `char*` parameters to the `add_item_consume` and `add_value_consume` functions and the `Any*` parameter to the `add_value_consume` function are consumed by the NVList. The caller may not access these data after they have been passed to these
functions because the NVList may copy them and destroy the originals immediately. The caller should use the NamedValue::value() operation to modify the value attribute of the underlying NamedValue, if desired.

- The remove function also calls CORBA::release on the removed NamedValue.

The following sections describe NVList member functions.

**CORBA::NVList::add**

**Synopsis**

Constructs a NamedValue object with an unnamed item, setting only the flags attribute.

**C++ Binding**

```cpp
CORBA::NamedValue_ptr CORBA::NVList::add (CORBA::Flags Flags);
```

**Argument**

**Flags**

Flags to determine argument passing. Valid values are:

- CORBA::ARG_IN
- CORBA::ARG_INOUT
- CORBA::ARG_OUT

**Description**

This member function constructs a NamedValue object with an unnamed item, setting only the flags attribute. The NamedValue object is added to the NVList object that the call was invoked upon.

The NVList object grows dynamically; your application does not need to track its size.

**Return Values**

If the function succeeds, the return value is a pointer to the newly created NamedValue object. The returned NamedValue object reference is owned by the NVList and should not be released.

If the member function does not succeed, a CORBA::NO_MEMORY exception is thrown.
See Also

CORBA::NVList::add
CORBA::NVList::add_item
CORBA::NVList::add_value
CORBA::NVList::count
CORBA::NVList::remove

CORBA::NVList::add_item

Synopsis

Constructs a NamedValue object, creating an empty value attribute and initializing the name and flags attributes.

C++ Binding

CORBA::NamedValue_ptr CORBA::NVList::add_item (const char * Name, CORBA::Flags Flags);

Arguments

Name

The name of the list item.

Flags

Flags to determine argument passing. Valid values are:

- CORBA::ARG_IN
- CORBA::ARG_INOUT
- CORBA::ARG_OUT

Description

This member function constructs a NamedValue object, creating an empty value attribute and initializing the name and flags attributes that pass in as parameters. The NamedValue object is added to the NVList object that the call was invoked upon.

The NVList object grows dynamically; your application does not need to track its size.
Return Values

If the function succeeds, the return value is a pointer to the newly created NamedValue object. The returned NamedValue object reference is owned by the NVList and should not be released.

If the member function does not succeed, an exception is thrown.

See Also

CORBA::NVList::add
CORBA::NVList::add_value
CORBA::NVList::count
CORBA::NVList::item
CORBA::NVList::remove

**CORBA::NVList::add_value**

Synopsis

Constructs a NamedValue object, initializing the name, value, and flags attribute.

C++ Binding

```cpp
CORBA::NamedValue_ptr CORBA::NVList::add_value (
    const char * Name,
    const CORBA::Any & Value,
    CORBA::Flags Flags);
```

Arguments

Name

The name of the list item.

Value

The value of the list item.

Flags

Flags to determine argument passing. Valid values are:

CORBA::ARG_IN
CORBA::ARG_INOUT
CORBA::ARG_OUT
Description
This member function constructs a NamedValue object, initializing the name, value, and flags attributes. The NamedValue object is added to the NVList object that the call was invoked upon.

The NVList object grows dynamically; your application does not need to track its size.

Return Values
If the function succeeds, the return value is a pointer to the newly created NamedValue object. The returned NamedValue object reference is owned by the NVList and should not be released.

If the member function does not succeed, an exception is raised.

See Also
CORBA::NVList::add
CORBA::NVList::add_item
CORBA::NVList::count
CORBA::NVList::item
CORBA::NVList::remove

CORBA::NVList::count

Synopsis
Retrieves the current number of items in the list.

C++ Binding

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

Arguments
None.

Description
This member function retrieves the current number of items in the list.

Return Values
If the function succeeds, the returned value is the number of items in the list. If the list has just been created, and no NamedValue objects have been added, this function returns 0 (zero).

If the function does not succeed, an exception is thrown.
See Also

CORBA::NVList::add
CORBA::NVList::add_item
CORBA::NVList::add_value
CORBA::NVList::item
CORBA::NVList::remove

CORBA::NVList::item

Synopsis

Retrieves a pointer to the NamedValue object, based on the index passed in.

C++ Binding

```cpp
CORBA::NamedValue_ptr CORBA::NVList::item (CORBA::ULong Index);
```

Argument

Index

The index into the NVList object. The indexing is zero-based.

Exception

If this function does not succeed, the BAD_PARAM exception is thrown.

Description

This member function retrieves a pointer to a NamedValue object, based on the index passed in. The function uses zero-based indexing.

Return Values

If the function succeeds, the return value is a pointer to the NamedValue object. The returned NamedValue object reference is owned by the NVList and should not be released.

See Also

CORBA::NVList::add
CORBA::NVList::add_item
CORBA::NVList::add_value
CORBA::NVList::count
CORBA::NVList::remove
**CORBA::NVList::remove**

**Synopsis**

Removes the item at the specified index, frees any associated memory, and reorders the remaining items on the list.

**C++ Binding**

```cpp
void CORBA::NVList::remove (CORBA::ULong Index);
```

**Argument**

`Index`

The index into the NVList object. The indexing is zero-based.

**Exception**

If this function does not succeed, the `BAD_PARAM` exception is thrown.

**Description**

This member function removes the item at the specified index, frees any associated memory, and reorders the remaining items on the list.

**Return Values**

None.

**See Also**

- `CORBA::NVList::add`
- `CORBA::NVList::add_item`
- `CORBA::NVList::add_value`
- `CORBA::NVList::count`
- `CORBA::NVList::item`
Object Member Functions

The rules in this section apply to the OMG IDL interface Object, which is the base of the OMG IDL interface hierarchy. Interface Object defines a normal CORBA object, not a pseudo-object. However, it is included here because it references other pseudo-objects.

In addition to other rules, all operation names in interface Object have leading underscores in the mapped C++ class. Also, the mapping for create_request is divided into three forms, corresponding to the usage styles described in the section Request Member Functions. The is_nil and release functions are provided in the CORBA namespace, as described in Object Member Functions.

The BEA Tuxedo software uses object reference operations that are defined by CORBA Revision 2.2. These operations depend only on type Object, so they can be expressed as regular functions within the CORBA namespace.

Note: Because the BEA Tuxedo software uses the POA and not the BOA, the deprecated get_implementation() member function is not visible; you will get a compile error if you attempt to reference it.

The mapping of these member functions to C++ is as follows:

class CORBA
{
    class Object
    {
        public:
            CORBA::Boolean _is_a(const char *)
            CORBA::Boolean _is_equivalent();
            CORBA::Boolean _nonexistent(Object_ptr);

        static Object_ptr _duplicate(Object_ptr obj);
        static Object_ptr _nil();
        InterfaceDef_ptr _get_interface();
        CORBA::ULong _hasa(CORBA::ULong);
        void _create_request(
            Context_ptr ctx,
            const char *operation,
            NVList_ptr arg_list,
            NamedValue_ptr result,
            Request_out request,
            Request_out request);
The following sections describe the Object member functions.

**CORBA::Object::create_request**

**Synopsis**

Creates a request with user-specified information.

**C++ Binding**

```cpp
void CORBA::Object::create_request(
    CORBA::Context_ptr Ctx,
    const char * Operation,
    CORBA::NVList_ptr Arg_list,
    CORBA::NamedValue_ptr Result,
    CORBA::ExceptionList_ptr Except_list,
    CORBA::ContextList_ptr Context_list,
    CORBA::Request_out Request,
    CORBA::Flags Req_flags);
```

**Arguments**

- **Ctx**
  
  The Context to be used for this request.
Object Member Functions

Operation
The operation name for this request.

Arg_list
The argument list for this request.

Result
The NamedValue reference where the return value of this request is to be stored after a successful invocation.

Except_list
The exception list for this request.

Context_list
The context list for this request.

Request
The newly created request reference.

Req_flags
Reserved for future use; the user must pass a value of zero.

Description
This member function creates a request that provides information on context, operation name, and other values (long form). To create a request with just the operation name supplied at the time of the call (short form), use the CORBA::Object::_request member function. The remainder of the information provided in the long form eventually needs to be supplied.

Return Values
None.

See Also
CORBA::Object::_request

CORBA::Object::_duplicate

Synopsis
Duplicates the Object object reference.
C++ Binding

```cpp
CORBA::Object_ptr CORBA::Object::_duplicate(
    Object_ptr Obj);
```

Argument

`obj`
The object reference to be duplicated.

Description

This member function duplicates the specified Object object reference (`Obj`). If the given object reference is nil, the `_duplicate` function returns a nil object reference. The object returned by this call should be freed using `CORBA::release`, or should be assigned to `CORBA::Object_var` for automatic destruction.

This function can throw CORBA system exceptions.

Return Values

Returns the duplicate object reference. If the specified object reference is nil, a nil object reference is returned.

Example

```cpp
CORBA::Object_ptr op = TP::create_object_reference(
    "IDL:Teller:1.0","MyTeller");
CORBA::Object_ptr dop = CORBA::Object::_duplicate(op);
```

**CORBA::Object::_get_interface**

Synopsis

Returns an interface definition for the Repository object.

C++ Binding

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

Arguments

None.
Description

Returns an interface definition for the Repository object.

Note: To use the Repository Interface API, define a macro before CORBA.h is included. For information about how to define a macro, see Creating CORBA Server Applications.

Return Values

InterfaceDef_ptr

**CORBA::Object:::_is_a**

Synopsis

Determines whether an object is of a certain interface.

C++ Binding

```cpp
CORBA::Boolean CORBA::Object:::_is_a(const char * interface_id);
```

Argument

`interface_id`

A string that denotes the interface repository ID.

Description

This member function is used to determine if an object is an instance of the interface that you specify in the `interface_id` parameter. It facilitates maintaining type-safety for object references over the scope of an ORB.

Return Values

Returns `TRUE` if the object is an instance of the specified type, or if the object is an ancestor of the “most derived” type of that object.

Example

```cpp
CORBA::Object_ptr op = TP::create_object_reference(  
    "IDL:Teller:1.0", "MyTeller");
CORBA::Boolean b = op->_is_a("IDL:Teller:1.0");
```
**CORBA::Object::_is_equivalent**

**Synopsis**
Determines if two object references are equivalent.

**C++ Binding**
```cpp
CORBA::Boolean CORBA::Object::_is_equivalent (
    CORBA::Object_ptr other_obj);
```

**Argument**
- **other_obj**
  The object reference for the other object, which is used for comparison with the target object.

**Exceptions**
Can throw a standard CORBA exception.

**Description**
This member function is used to determine if two object references are equivalent, so far as the ORB can easily determine. It returns `TRUE` if your object reference is equivalent to the object reference you pass as a parameter. If two object references are identical, they are equivalent. Two different object references that refer to the same object are also equivalent.

**Return Values**
- **Returns** `TRUE` if the target object reference is known to be equivalent to the other object reference passed as a parameter; otherwise, it returns `FALSE`.

**Example**
```cpp
CORBA::Object_ptr op = TP::create_object_reference(
    "IDL:Teller:1.0", "MyTeller");
CORBA::Object_ptr dop = CORBA::Object::_duplicate(op);
CORBA::Boolean b = op->_is_equivalent(dop);
```

---

**CORBA::Object::_nil**

**Synopsis**
Returns a reference to a nil object.
Object Member Functions

C++ Binding

CORBA::Object_ptr CORBA::Object::_nil();

Arguments
None.

Description
This member function returns a nil object reference. To test whether a given object is nil, use the appropriate CORBA::is_nil member function (see the section CORBA::release). Calling the CORBA::is_nil routine on any _nil member function always yields CORBA_TRUE.

Return Values
Returns a nil object reference.

Example

CORBA::Object_ptr op = CORBA::Object::_nil();

CORBA::Object::_non_existent

Synopsis
May be used to determine if an object has been destroyed.

C++ Binding

CORBA::Boolean CORBA::Object::_non_existent();

Arguments
None.

Description
This member function may be used to determine if an object has been destroyed. It does this without invoking any application-level operation on the object, and so will never affect the object itself.

Return Values
Returns CORBA_TRUE (rather than raising CORBA::OBJECT_NOT_EXIST) if the ORB knows authoritatively that the object does not exist; otherwise, it returns CORBA_FALSE.
CORBA::Object::_request

Synopsis

Creates a request specifying the operation name.

C++ Binding

```
CORBA::Request_ptr CORBA::Object::_request (
    const char  *      Operation);
```

Argument

Operation

The name of the operation for this request.

Description

This member function creates a request specifying the operation name. All other information, such as arguments and results, must be populated using `CORBA::Request` member functions.

Return Values

If the member function succeeds, the return value is a pointer to the newly created request.

If the member function does not succeed, an exception is thrown.

See Also

`CORBA::Object::_create_request`

CORBA Member Functions

This section describes the Object and Pseudo-Object Reference member functions.

The mapping of these member functions to C++ is as follows:

```c
class CORBA {
    void release(Object_ptr);
    void release(Environment_ptr);
    void release(NamedValue_ptr);
    void release(NVList_ptr);
    void release(Request_ptr);
    void release(Context_ptr);
    void release(TypeCode_ptr);
}
```
void release(POA_ptr);
void release(ORB_ptr);
void release(ExceptionList_ptr);
void release(ContextList_ptr);

Boolean is_nil(Object_ptr);
Boolean is_nil(Environment_ptr);
Boolean is_nil(NamedValue_ptr);
Boolean is_nil(NVList_ptr);
Boolean is_nil(Request_ptr);
Boolean is_nil(Context_ptr);
Boolean is_nil(TypeCode_ptr);
Boolean is_nil(POA_ptr);
Boolean is_nil(ORB_ptr);
Boolean is_nil(ExceptionList_ptr);
Boolean isNil(ContextList_ptr);

hash(maximum);

resolve_initial_references(identifier);
...
};

CORBA::release

Synopsis
Allows allocated resources to be released for the specified object type.

C++ Binding
void CORBA::release(spec_object_type obj);

Argument
obj
The object reference that the caller will no longer access. The specified object type must be one of the types listed in the section CORBA Member Functions.
Description

This member function indicates that the caller will no longer access the reference so that associated resources may be deallocated. If the specified object reference is nil, the release operation does nothing. If the ORB instance release is the last reference to the ORB, then the ORB will be shut down prior to its destruction. This is the same as calling ORB_shutdown prior to calling CORBA::release. This only applies to the release member function called on the ORB.

This member function may not throw CORBA exceptions.

Return Values

None.

Example

CORBA::Object_ptr op = TP::create_object_reference("IDL:Teller:1.0", "MyTeller");
CORBA::release(op);

CORBA::is_nil

Synopsis

Determines if an object exists for the specified object type.

C++ Binding

CORBA::Boolean CORBA::is_nil(spec_object_type obj);

Argument

obj

The object reference. The specified object type must be one of the types listed in the section CORBA Member Functions.

Description

This member function is used to determine if a specified object reference is nil. It returns TRUE if the object reference contains the special value for a nil object reference as defined by the ORB.

This operation may not throw CORBA exceptions.

Return Values

Returns TRUE if the specified object is nil; otherwise, returns FALSE.
Example

```
CORBA::Object_ptr op = TP::create_object_reference(
    "IDL:Teller:1.0", "MyTeller");
CORBA::Boolean b = CORBA::is_nil(op);
```

**CORBA::hash**

**Synopsis**

Provides indirect access to object references using identifiers internal to the ORB.

**C++ Binding**

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

**Argument**

<table>
<thead>
<tr>
<th>maximum</th>
</tr>
</thead>
<tbody>
<tr>
<td>Specifies an upper bound on the hash value returned by the ORB.</td>
</tr>
</tbody>
</table>

**Description**

Object references are associated with ORB-internal identifiers that may indirectly be accessed by applications using the `hash()` operation. The value of this identifier does not change during the lifetime of the object reference, and so neither will any hash function of that identifier.

The value of this operation is not guaranteed to be unique; that is, another object reference may return the same hash value. However, if two object references hash differently, applications can determine that the two object references are not identical.

The `maximum` parameter to the `hash` operation specifies an upper bound on the hash value returned by the ORB. The lower bound of that value is zero. Since a typical use of this feature is to construct and access a collision-chained hash table of object references, the more randomly distributed the values are within that range, and the less expensive those values are to compute, the better.

**Return Values**

None.
CORBA::resolve_initial_references

Synopsis
Returns an initial object reference corresponding to an identifier string.

C++ Binding
CORBA::Object_ptr CORBA::resolve_initial_references(
    const CORBA::char *identifier);

Argument
identifier
String identifying the object whose reference is required.

Exception
InvalidName

Description
Returns an initial object reference corresponding to an identifier string. Valid identifiers are "RootPOA" and "POACurrent".

Note: This function is supported only for a joint client/server.

Return Values
Returns a CORBA::Object_ptr.

Example
CORBA::ORB_ptr orb = CORBA::ORB_init(argc, argv);
CORBA::Object_ptr pfobj =
    orb->resolve_initial_references("RootPOA");
PortableServer::POA_ptr rootPOA;
rootPOA = PortableServer::POA::narrow(pfobj);

ORB Member Functions
The ORB member functions constitute the programming interface to the Object Request Broker.
The mapping of the ORB member functions to C++ is as follows:
ORB Member Functions

```cpp
class CORBA
{
class ORB
{
public:
    char *object_to_string(Object_ptr);
    Object_ptr string_to_object(const char *);
    void create_list(Long, NVList_out);
    void create_operation_list(operationDef_ptr, NVList_out);
    void create_named_value(NamedValue_out);
    void create_exception_list(ExceptionList_out);
    void create_context_list(ContextList_out);
    void get_default_context(Context_out);
    void create_environment(Environment_out);
    void send_multiple_requests_oneway(const requestSeq&);
    void send_multiple_requests_deferred(const requestSeq&);
    Boolean poll_next_response();
    void get_next_response(Request_out);
    Boolean work_pending();
    void perform_work();
    void create_policy (in PolicyType type, in any val);
    // Extension
    void destroy();
    // Extensions to support sharing context between threads
    void Ctx get_ctx() = 0;
    void set_ctx(Ctx) = 0;
    void clear_ctx() = 0;
    // Thread extensions
    void inform_thread_exit(TID) = 0;
}; //ORB
}; // CORBA
```

Thread-related Operations:

To support single-threaded ORBs, as well as multithreaded ORBs that run multithread-unaware code, two operations (perform_work and work_pending) are included in the ORB interface. These operations can be used by single-threaded and multithreaded applications. An application that is a pure ORB client would not need to use these operations.
To support multithreaded server applications, four operations (get_ctx, set_ctx, clear_ctx, and inform_thread_exit) are included as extensions to the ORB interface.

The following sections describe the ORB member functions.

**CORBA::ORB::clear_ctx**

**Synopsis**

Indicates that a context is no longer required by this thread. This method supports the development of a multithreaded server application.

**C++ Binding**

```cpp
void clear_ctx()
```

**Parameters**

None.

**Return Value**

None.

**Description**

This method is called by an application-managed thread after the thread has finished using the context. The method removes the association between that thread and a context.

**Note:** Do not call the clear_ctx method from within a thread that is managed by the BEA Tuxedo system. The BEA Tuxedo system performs the appropriate context propagation and cleanup automatically for the threads it manages. If this method is called on a thread managed by the BEA Tuxedo system, the BAD_PARAM exception is thrown.

**Example**

```cpp
TP::orb()->clear_ctx();
```

**See Also**

CORBA::ORB::get_ctx
CORBA::ORB::set_ctx
**CORBA::ORB::create_context_list**

**Synopsis**

Creates and returns a list of contexts.

**C++ Binding**

```cpp
void CORBA::ORB::create_context_list(
    CORBA::ContextList_out List);
```

**Argument**

List

Receives a reference to the newly created context list.

**Description**

This member function creates and returns a list of context strings that must be supplied with the Request operation in a form that may be used in the Dynamic Invocation Interface (DII). When no longer needed, this list must be freed using the CORBA::release member function.

**Return Values**

None.

**CORBA::ORB::create_environment**

**Synopsis**

Creates an environment.

**C++ Binding**

```cpp
void CORBA::ORB::create_environment (
    CORBA::Environment_out New_env);
```

**Argument**

New_env

Receives a reference to the newly created environment.

**Description**

This member function creates an environment.
Return Values
None.

See Also
CORBA::NVList::add
CORBA::NVList::add_item
CORBA::NVList::add_value
CORBA::release

**CORBA::ORB::create_exception_list**

Synopsis
Returns a list of exceptions.

C++ Binding
```cpp
void CORBA::ORB::create_exception_list(  
    CORBA::ExceptionList_out   List);
```

Argument
List
Receives a reference to the newly created exception list.

Description
This member function creates and returns a list of exceptions in a form that may be used in the Dynamic Invocation Interface (DII). When no longer needed, this list must be freed using the CORBA::release member function.

Return Values
None.

**CORBA::ORB::create_list**

Synopsis
Creates and returns an NVList object reference.
ORB Member Functions

C++ Binding

```cpp
void CORBA::ORB::create_list (
    CORBA::Long               NumItem,
    CORBA::NVList_out          List);
```

Arguments

- **NumItem**
  
  The number of elements to preallocate in the newly created list.

- **List**
  
  Receives the newly created list.

Description

This member function creates a list, preallocating a specified number of items. List items may be sequentially added to the list using the `CORBA::NVList::add_item` member function. When no longer needed, this list must be freed using the `CORBA::release` member function.

Return Values

None.

See Also

- `CORBA::NVList::add`
- `CORBA::NVList::add_item`
- `CORBA::NVList::add_value`
- `CORBA::release`

**CORBA::ORB::create_named_value**

Synopsis

Creates a NamedValue object reference.

C++ Binding

```cpp
void CORBA::ORB::create_named_value (
    NameValue_out         NewNamedVal);
```
Argument

NewNamedVal
A reference to the newly created NamedValue object.

Description
This member function creates a NamedValue object. Its intended use is for the result argument of a request that needs a NamedValue object. The extra steps of creating an NVList object are avoided by calling this member function.

When no longer needed, the NamedValue object must be freed using the CORBA::release member function.

Return Values
None.

See Also
CORBA::NVList::add
CORBA::NVList::add_item
CORBA::NVList::add_value
CORBA::release

CORBA::ORB::create_operation_list

Synopsis
Creates and returns a list of the arguments of a specified operation.

C++ Binding

```cpp
void CORBA::ORB::create_operation_list (
    CORBA::OperationDef_ptr Oper,
    CORBA::NVList_out List);
```

Arguments

Oper
The operation definition for which the list is being created.

List
Receives a reference to the newly created arguments list.
Description

This member function creates and returns a list of the arguments of a specified operation, in a form that may be used with the Dynamic Invocation Interface (DII). When no longer needed, this list must be freed using the `CORBA::release` member function.

Return Values

None.

See Also

`CORBA::OBB::create_list`
`CORBA::NVList::add`
`CORBA::NVList::add_item`
`CORBA::NVList::add_value`
`CORBA::release`

**CORBA::ORB::create_policy**

Synopsis

Creates new instances of policy objects of a specific type with specified initial state.

C++ Binding

```cpp
void CORBA::ORB::create_policy (  
    in PolicyType type,  
    in any val);  
```

Arguments

type

`BiDirPolicy::BIDIRECTIONAL_POLICY_TYPE` is the only `PolicyType` value supported for BEA WebLogic Enterprise version 4.2.

val

The only `val` value supported for BEA WebLogic Enterprise V4.2 is `BiDirPolicy::BidirectionalPolicyValue`.

Exceptions

`PolicyError`

This exception is raised to indicate problems with the parameter values passed to the `ORB::create_policy` operation. The specific exception and reasons are as follows:
This operation can be invoked to create new instances of policy objects of a specific type with specified initial state. If `create_policy` fails to instantiate a new Policy object due to its inability to interpret the requested type and content of the policy, it raises the Policy Error exception with the appropriate reason. (See Exceptions below.)

The `BidirectionalPolicy` argument is provided for remote clients using callbacks because remote clients use IIOP. It is not used for native clients using callbacks or for BEA Tuxedo servers because machines inside a BEA Tuxedo domain communicate differently.

Before GIOP 1.2, bidirectional policy was not available as a choice in IIOP (which uses TCP/IP). Connections in GIOP 1.0 and 1.1 were one way (that is, a request flowed from a client to a server); only responses flowed from the server back to the client. If the server wanted to make a request back to the client machine (say for a callback), the server machine had to establish another one-way connection. (Be advised that “connections” in this sense mean operating system resources, not physically different wires or communication paths. A connection uses resources, so minimizing connections is desirable.)

Since this release of the BEA Tuxedo C++ software supports GIOP 1.2, it supports reuse of the TCP/IP connection for both incoming and outgoing requests. Reusing connections saves resources when a remote client sends callback references to a BEA Tuxedo domain. The joint client/server uses a connection to send a request to a BEA Tuxedo domain; that connection can

<table>
<thead>
<tr>
<th>Exception</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>BAD_POLICY</td>
<td>The requested Policy is not understood by the ORB.</td>
</tr>
<tr>
<td>UNSUPPORTED_POLICY</td>
<td>The requested Policy is understood to be valid by the ORB, but is not currently supported.</td>
</tr>
<tr>
<td>BAD_POLICY_TYPE</td>
<td>The type of the value requested for the Policy is not valid for that PolicyType.</td>
</tr>
<tr>
<td>BAD_POLICY_VALUE</td>
<td>The value requested for the Policy is of a valid type, but is not within the valid range for that type.</td>
</tr>
<tr>
<td>UNSUPPORTED_POLICY_VALUE</td>
<td>The value requested for the Policy is of a valid type and within the valid range for that type, but this valid value is not currently supported.</td>
</tr>
</tbody>
</table>
be reused for the callback request. If the connection is not reused, the callback request must establish another connection.

Allowing reuse of a connection is a choice of the ORB/POA that creates callback object references. The server for those object references (usually the creator of the references, especially in the callback case) might choose not to allow reuse for security considerations (that is, the outgoing connection [a client request from this machine to a remote server] may not need security because the remote server does not require it, but the callback server on this machine might require security). Since security is established partly on a connection basis, the incoming security can be established only if a separate connection is used. If the remote server requires security, and if that security involves a mutual authentication, the local server usually feels safe in allowing reuse of the connection.

Since the choice of connection reuse is at the server end, whenever a process acts as a server—in this case a joint client/server—and creates object references, it must inform the ORB that it is willing to reuse connections. The process does this by setting a policy on the POA that creates the object references. The default policy is to not allow reuse (that is, if you do not supply a policy object for reuse, the POA does not allow reuse).

This default allows for backward compatibility with code written before CORBA version 2.3. Such code did not know that reuse was possible so it did not have to take into consideration the security implications of reuse. Thus, that unchanged code should continue to disallow reuse until the user considers security and explicitly makes a decision to the contrary.

To allow reuse, you use the `create_policy` operation to create a policy object that allows reuse, and use that policy object as part of the list of policies for POA creation.

Return Values
None.

Example

```c++
#include <BiDirPolicy_c.h>
BiDirPolicy::BidirectionalPolicy_var bd_policy;
CORBA::Any allow_reuse;
allow_reuse <<= BiDirPolicy::BOTH;
CORBA::Policy_var generic_policy =
    orb->create_policy( BiDirPolicy::BIDIRECTIONAL_POLICY_TYPE,
                        allow_reuse );
```
bd_policy = BiDirPolicy::BidirectionalPolicy::narrow(
    generic_policy);

In the above example, the bd_policy would then be placed in the PolicyList passed to the 
create_poa operation.

**CORBA::ORB::destroy**

**Synopsis**

Destroys the specified ORB.

**C++ Binding**

```cpp
void destroy();
```

**Parameter**

None.

**Return Value**

None.

**Description**

Use this method to destroy an ORB so that the resources associated with that ORB can be 
reclaimed. Once an ORB has been destroyed, another invocation on the ORB_init method with 
the same ORB ID returns a reference to a newly constructed ORB. If an application invokes the 
ORB::destroy method from a thread that is currently servicing an invocation, the BEA Tuxedo 
system raises the BAD_INV_ORDER system exception with the OMG minor code 3, because 
blocking would result in a deadlock.

**Example**

```cpp
pOrb->destroy();
```

**CORBA::ORB::get_ctx**

**Synopsis**

Retrieves the context associated with the current thread. This method supports the development 
of a multithreaded server application.
ORB Member Functions

C++ Binding

CORBA::ORB::Ctx get_ctx()

Arguments

None.

Return Value

CORBA::ORB::Ctx

The context associated with this thread.

Description

Use this method to retrieve the context associated with the current thread. This context can then be used to initialize other threads that the application creates and manages.

When an object creates a thread, the object invokes this operation on the ORB to obtain system context information that the object can pass on to the thread. This operation must be called from a thread that already has a context. For example, the thread in which a method was dispatched will already have a context.

Example

thread.context = TP::orb()->get_ctx();

See Also

CORBA::ORB::set_ctx
CORBA::ORB::clear_ctx

CORBA::ORB::get_default_context

Synopsis

Returns a reference to the default context.

C++ Binding

void CORBA::ORB::get_default_context ( 
    CORBA::Context_out   ContextObj);
Argument

ContextObj

The reference to the default context.

Description

This member function returns a reference to the default context. When no longer needed, this context reference must be freed using the CORBA::release member function.

Return Values

None.

See Also

CORBA::Context::get_one_value
CORBA::Context::get_values

CORBA::ORB::get_next_response

Synopsis

Determines and reports the next deferred synchronous request that completes.

C++ Binding

```cpp
void CORBA::ORB::get_next_response (  
    CORBA::Request_out RequestObj);  
```

Argument

RequestObj

The reference to the next completed request.

Description

This member function returns a reference to the next request that completes. If no requests have completed, the function waits for a request to complete. This member function returns the next request on the queue, in contrast to the CORBA::Request::get_response member function, which waits for a particular request to complete. When no longer needed, this request must be freed using the CORBA::release member function.

Return Values

None.
See Also
- CORBA::ORB::poll_next_response
- CORBA::Request::get_response

**CORBA::ORB::inform_thread_exit**

**Synopsis**
Informs the BEA Tuxedo system that resources associated with an application-managed thread can be released. This method supports the development of a multithreaded server application.

**C++ Binding**
```cpp
void CORBA::ORB::inform_thread_exit(CORBA::TID threadId)
```

**Parameter**
- `threadId`
The logical thread ID of the application-managed thread being deleted.

**Return Value**
None.

**Description**
This method informs the BEA Tuxedo system about the following conditions:

- The specified application-managed thread is no longer used by a servant implementation.
- Any resources associated with the thread should be released.

**Note:** You should only call this operation on threads that the application creates and manages. Do not invoke this method when specifying a dispatch thread that is managed by the BEA Tuxedo system.

**Example**
```cpp
pOrb->inform_thread_exit(thread.threadId);
```

**CORBA::ORB::list_initial_services**

**Synopsis**
Determines which objects have references available via the initial references mechanism.
C++ Binding

typedef string ObjectId;
typedef sequence ObjectId ObjectIdList;
ObjectIdList list_initial_services ();

Argument

ObjectId
The object ID.

list_initial_services ()
Defines the object type.

Description

This operation is used by applications to determine which objects have references available via the initial references mechanism. This operation returns an ObjectIdList, which is a sequence of ObjectIds. ObjectIds are typed as strings.

Each object, which may need to be made available at initialization time, is allocated a string value to represent it. In addition to defining the ID, the type of object being returned must be defined, that is, InterfaceRepository returns an object of type Repository, and NameService returns a CosNamingContext object.

Return Values

Sequence of ObjectIds.

See Also

CORBA::ORB::resolve_initial_references

CORBA::ORB::object_to_string

Synopsis

Produces a string representation of an object reference.

C++ Binding

char * CORBA::ORB::object_to_string (CORBA::Object_ptr ObjRef);
Argument

ObjRef

The object reference to represent as a string.

Description

This member function produces a string representation of an object reference. The calling program must use the \texttt{CORBA::string\_free} member function to free the string memory after it is no longer needed.

Return Values

The string representing the specified object reference.

Example

\begin{verbatim}
CORBA::Object\_ptr op = TP::create\_object\_reference(
    "IDL:Teller:1.0", "MyTeller");
char* objstr = TP::orb\()->object\_to\_string(op);
\end{verbatim}

See Also

\texttt{CORBA::ORB::string\_to\_object}
\texttt{CORBA::string\_free}

\textbf{CORBA::ORB::perform\_work}

Synopsis

Allows the ORB to perform server-related work.

C++ Binding

\begin{verbatim}
void CORBA::ORB::perform\_work ();
\end{verbatim}

Arguments

None.

Exceptions

Once the ORB has shut down, a call to \texttt{work\_pending} and \texttt{perform\_work()} raises the \texttt{BAD\_INV\_ORDER} exception. An application can detect this exception to determine when to terminate a polling loop.
Description

If called by the main thread, this operation allows the ORB to perform server-related work. Otherwise, it does nothing.

The work_pending() and perform_work() operations can be used to write a simple polling loop that multiplexes the main thread among the ORB and other activities. Such a loop would most likely be needed in a single-threaded server. A multithreaded server would need a polling loop only if there were both ORB and other code that required use of the main thread. See the example below for such a polling loop.

Return Values

None.

See Also

CORBA::ORB::work_pending

Example

The following is an example of a polling loop:

```cpp
// C++
for (;;) {
    if (orb->work_pending()) {
        orb->perform_work();
    }
    // do other things
    // sleep?
}
```

**CORBA::ORB::poll_next_response**

Synopsis

Determines whether a completed request is outstanding.

C++ Binding

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

Arguments

None.
Description

This member function reports on whether there is an outstanding (pending) completed request; it does not remove the request. If a completed request is outstanding, the next call to the `CORBA::ORB::get_next_response` member function is guaranteed to return a request without waiting. If there are no completed requests outstanding, the `CORBA::ORB::poll_next_response` member function returns without waiting (blocking).

Return Values

If a completed request is outstanding, the function returns `CORBA_TRUE`.
If no completed request is outstanding, the function returns `CORBA_FALSE`.

See Also

`CORBA::ORB::get_next_response`

`CORBA::ORB::resolve_initial_references`

Synopsis

Obtains object references for initial services.

C++ Binding

```cpp
Object resolve_initial_references ( in ObjectId identifier )
    raises (InvalidName);

exception InvalidName {};
```

Augument

`identifier`

String that identifies the object whose reference is required.

Description

This operation is used by applications to obtain object references for initial services. The interface differs from the Naming Service’s resolve in that `ObjectId` (a string) replaces the more complex Naming Service construct (a sequence of structures containing string pairs for the components of the name). This simplification reduces the namespace to one context.

`ObjectId` are strings that identify the object whose reference is required. To maintain the simplicity of the interface for obtaining initial references, only a limited set of objects are expected to have their references found via this means. Unlike the ORB identifiers, the `ObjectId`
name space requires careful management. To achieve this, the OMG may, in the future, define which services are required by applications through this interface and specify names for those services.

Currently, reserved ObjectIds are RootPOA, POACurrent, InterfaceRepository, NameService, TradingService, SecurityCurrent, TransactionCurrent, and DynAnyFactory.

The application is responsible for narrowing the object reference returned from `resolve_initial_references` to the type that was requested in the ObjectID. For example, for `InterfaceRepository` the object returned would be narrowed to `Repository` type.

Return Values

Object references for initial services.

See Also

`CORBA::ORB::list_initial_services`

**CORBA::ORB::send_multiple_requests_deferred**

Synopsis

Sends a sequence of deferred synchronous requests.

C++ Binding

```cpp
void CORBA::ORB::send_multiple_requests_deferred (  
const CORBA::ORB::RequestSeq & Reqs);
```

Argument

`Reqs`

The sequence of requests to be sent. For more information about how to populate the sequence with request references, see `CORBA::ORB::RequestSeq` in the section Usage.

Description

This member function sends out a sequence of requests and returns control to the caller without waiting for the operation to complete. The caller uses `CORBA::ORB::poll_next_response`, `CORBA::ORB::get_next_response`, or `CORBA::Request::get_response` or all three to determine if the operation has completed and if the output arguments have been updated.
ORB Member Functions

Return Values
None.

See Also
CORBA::Request::get_response
CORBA::ORB::get_next_response
CORBA::ORB::send_multiple_requests_oneway

Synopsis
Sends a sequence of one-way, deferred synchronous requests.

C++ Binding

```cpp
void CORBA::ORB::send_multiple_requests_oneway (const CORBA::RequestSeq & Reqs);
```

Argument

**Reqs**
The sequence of requests to be sent. For more information about how to populate the sequence with request references, see `CORBA::ORB::RequestSeq` in the section Usage.

Description
This member function sends out a sequence of requests and returns control to the caller without waiting for the operation to complete. The caller neither intends to wait for a response nor expects any output arguments to be updated.

Return Values
None.

See Also
CORBA::ORB::send_multiple_requests_deferred
**CORBA::ORB::set_ctx**

**Synopsis**
Sets the context for the current thread. This method supports the development of a multithreaded server application.

**C++ Binding**

```cpp
void set_ctx(CORBA::ORB::Ctx aContext)
```

**Parameter**

`aContext`

The context to be associated with this thread.

**Return Value**

None.

**Description**
This method sets the context for the current application-managed thread. The context parameter provided must have been obtained in a previously-executed thread that is managed by the BEA Tuxedo system or in an application-managed thread that has already been initialized.

**Note:** Do not call the `set_ctx` method in a thread that is managed by the BEA Tuxedo system. The BEA Tuxedo system performs the appropriate context propagation automatically for the threads it manages. If your application calls this method on a thread managed by the BEA Tuxedo system, the BAD_PARAM exception is thrown.

**Example**

```cpp
TP::orb()->set_ctx(thread->context);
```

**See Also**

CORBA::ORB::get_ctx()
CORBA::ORB::clear_ctx()

**CORBA::ORB::string_to_object**

**Synopsis**
Converts a string produced by `CORBA::ORB::object_to_string` operation and returns the corresponding object reference.
C++ Binding

```cpp
Object string_to_object ( in string str );
```

**Argument**

*str*

String produced by the CORBA::ORB::object_to_string operation.

**Description**

This operation is used by applications to convert a string produced by CORBA::ORB::object_to_string operation and returns the corresponding object reference.

To guarantee that an ORB will understand the string form of an object reference, that ORB’s object_to_string operation must be used to produce the string. The string_to_object operation allows URLs in the IOR, corbaloc, corbalocs, and corbanames formats to be converted into object references. If a conversion fails, the string_to_object operation raises the BAD_PARAM standard exception with one of the following minor codes:

- BadSchemeName
- BadAddress
- BadSchemeSpecificPart

For all conforming ORBs, if `obj` is a valid reference to an object, then `string_to_object(object_to_string(obj))` will return a valid reference to the same object, if the two operations are performed on the same ORB. For all conforming ORB’s supporting IOP, this remains true even if the two operations are performed on different ORBs.

**Return Value**

Returns an object reference.

**See Also**

CORBA::ORB::object_to_string

---

**CORBA::ORB::work_pending**

**Synopsis**

Returns an indication of whether the ORB needs the main thread to perform server-related work.

C++ Binding

```cpp
CORBA::boolean CORBA::ORB::work_pending ();
```
Arguments
None.

Description
This operation returns an indication of whether the ORB needs the main thread to perform server-related work.

Return Values
A result of `TRUE` indicates that the ORB needs the main thread to perform server-related work, and a result of `FALSE` indicates that the ORB does not need the main thread.

See Also
CORBA::ORB::perform_work

ORB Initialization Member Function

The mapping of this member function to C++ is as follows:

```cpp
class CORBA {
    static CORBA::ORB_ptr ORB_init(int& argc, char** argv,
        const char* orb_identifier = 0,
        const char* -ORBport nnn);

    <appl-name> [-ORBid {BEA_IIOAP | BEA_TOBJ} \n
        [-ORBInitRef <ObjectID>=<ObjectURL> [*]]
        [-ORBDefaultInitRef <ObjectURL>]
        [-ORBport port-number] \n
        [-ORBsecurePort port-number] \n
        [-ORBminCrypto {0 | 40 | 56 | 128}] \n
        [-ORBmaxCrypto {0 | 40 | 56 | 128}] \n
        [-ORBmutualAuth] \n
        [-ORBpeerValidate {detect | warn | none}] \n
        [appl-options]
    }

};
```
**CORBA::ORB_init**

**Synopsis**
Initializes operations for an ORB.

**C++ Binding**

```cpp
static CORBA::ORB_ptr ORB_init(int& argc, char** argv,
const char* orb_identifier = 0);
```

**Arguments**

- `argc`
  The number of strings in `argv`.

- `argv`
  This argument is defined as an unbound array of strings (`char **`) and the number of strings in the array is passed in the `argc` parameter.

- `orb_identifier`
  If the `orb_identifier` parameter is supplied, "BEA_IIOP" explicitly specifies a remote client and "BEA_TOBJ" explicitly specifies a native client, as defined in the section `Tobj_Bootstrap`.

**Description**

This member function initializes operations for an ORB and returns a pointer to the ORB. When your program is done with the ORB, use the `CORBA::release` member function to free the resources allocated for the ORB pointer returned from `CORBA::ORB_ptr ORB_init`.

The ORB returned has been initialized with two pieces of information to determine how it will operate: client type (remote or native) and server port number. The client type can be specified in the `orb_identifier` argument, in the `argv` argument, or in the system registry. The server port number can be specified in the `argv` argument.

The arguments `argc` and `argv` are typically the same parameters that were passed to the main program. As specified by C++, these parameters contain string tokens from the command line that started the client. The two ORB options can be specified on the command line, each using a pair of tokens, as shown in examples below.

**Client Type**

The `ORB_init` function determines the client type of the ORB by the following steps.
1. If the orb_identifier argument is present, ORB_init determines the client type, either native or remote, if the string is "BEA_IIOP" or "BEA_TOBJ", respectively. If an orb_identifier string is present, all -ORBid parameters in the argv are ignored (removed).

2. If orb_identifier is not present or is explicitly zero, ORB_init looks at the entries in argc/argv. If argv contains an entry with "-ORBid", the next entry should be either "BEA_IIOP" or "BEA_TOBJ", again specifying remote or native. This pair of entries occurs if the command line contains either "-ORBid BEA_IIOP" or "-ORBid BEA_TOBJ".

3. If no client type is specified in argc/argv, ORB_init uses the default client type from the system registry (BEA_IIOP or BEA_TOBJ). The system registry was initialized at the time BEA Tuxedo was installed.

Server Port

In the case of a BEA Tuxedo remote joint client/server, in order to support IIOP, by definition, the object references created for the server part must contain a host and port. For transient object references, any port is sufficient and can be obtained by the ORB dynamically, but this is not sufficient for persistent object references. Persistent references must be served on the same port after the ORB restarts, that is, the ORB must be prepared to accept requests on the same port with which it created the object reference. Thus, there must be some way to configure the ORB to use a particular port.

Typically, a system administrator assigns the port number for the client from the “user” range of port numbers rather from the dynamic range. This keeps the joint client/servers from using conflicting ports.

To determine port number, ORB_init searches the argv parameter for the token "-ORBport" and a following numeric token. For example, if the client executable is named sherry, the command line might specify that the server port should be 937 as follows:

```
sherry -ORBport 937
```

ARGV Parameter Considerations

For C++, the order of consumption of argv parameters may be significant to an application. To ensure that applications are not required to handle argv parameters they do not recognize, the ORB initialization function must be called before the remainder of the parameters are consumed. Therefore, after the ORB_init call, the argv and argc parameters have been modified to remove the ORB understood arguments. It is important to note that the ORB_init function can only reorder or remove references to parameters from the argv list. This restriction is made to avoid potential memory management problems caused by trying to free parts of the argv list or
extending the argv list of parameters. This is why argv is passed as a char** and not as a char**&.

**Note:** Use the CORBA::release member function to free the resources allocated for the pointer returned from CORBA::ORB_init.

Return Value

A pointer to a CORBA::ORB.

Exceptions

None.

**ORB**

**Synopsis**

Configures applications based on the BEA Tuxedo CORBA C++ ORB to access or provide BEA Tuxedo CORBA objects.

**Syntax**

```
<appl-name> [-ORBid {BEA_IIOP | BEA_TOBJ} \ 
             [-ORBInitRef <ObjectID>=<ObjectURL> [*]] \ 
             [-ORBDefaultInitRef <ObjectURL>] \ 
             [-ORBport port-number] \ 
             [-ORBsecurePort port-number] \ 
             [-ORBminCrypto {0 | 40 | 56 | 128}] \ 
             [-ORBmaxCrypto {0 | 40 | 56 | 128}] \ 
             [-ORBmutualAuth] \ 
             [-ORBpeerValidate {detect | warn | none}] \ 
             [appl-options]
```

**Description**

The BEA Tuxedo CORBA C++ ORB is a BEA Tuxedo-supplied library that enables the development of CORBA-based applications used to access or provide BEA Tuxedo objects using IIOP or IIOP-SSL. The ORB command-line options allow for customization.
Parameters

[-ORBid {BEA_IIOOP | BEA_TOBJ}]

The value BEA_IIOOP explicitly specifies that the ORB be configured to support either a client or a server environment that communicates over the IIOP or IIOP-SSL protocol. The value BEA_TOBJ explicitly specifies that the ORB be configured to support the native client environment that can only communicate over the TGIOP protocol within a BEA Tuxedo domain.

If not specified, the ORB will detect the environment in which it is deployed and configure itself for use in that environment.

[-ORBInitRef ObjectId=ObjectURL]

The ORB initial reference argument, -ORBInitRef, allows specification of an arbitrary object reference for an initial service. ObjectId represents the well-known object ID for a service that is defined in the CORBA specification. This mechanism allows an ORB to be configured with new initial service Object IDs that were not defined when the ORB was installed. ObjectURL can be any of the URL schemes supported by the CORBA::ORB::string_to_object operation as defined in CORBA specification. If a URL is syntactically malformed or can be determined to be invalid in an implementation-defined manner, CORBA::ORB_init will raise the CORBA::BAD_PARAM standard exception listed in Table 14-1.

Table 14-1 Minor Codes for CORBA::BAD_PARAM Standard Exception

<table>
<thead>
<tr>
<th>Minor Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BadSchemeName</td>
<td>The specified scheme is recognized by the ORB implementation. Only the schemes IOR, corbaloc, corbalocs, and corbaname are supported.</td>
</tr>
<tr>
<td>BadAddress</td>
<td>The format of the address is not recognized by the ORB implementation. Host names must be specified according to DNS or as class C IP addresses in dot-separated form.</td>
</tr>
<tr>
<td>BadSchemeSpecificPart</td>
<td>The format of the address is not recognized by the ORB implementation. Host names must be specified according to DNS or as class C IP addresses in dot-separated form.</td>
</tr>
<tr>
<td>BadSchemeSpecificPart</td>
<td>The scheme specific part of the URL is improperly formatted for the specified scheme.</td>
</tr>
</tbody>
</table>
The ORB default initial reference argument, -ORBDefaultInitRef, assists in the resolution of initial references not explicitly specified with -ORBInitRef. This argument provides functionality similar to that of the list of IIOP Listeners address that is provided to the current Tobj_Bootstrap object.

Unlike the -ORBInitRef argument, -ORBDefaultInitRef requires a URL that, after appending a slash '/' character and a stringified object key, forms a new URL to identify an initial object reference. For example, if the following was specified as the default initial reference argument:

-ORBDefaultInitRef corbaloc:555objs.com

A call to ORB:resolve_initial_references("NotificationService") to obtain the initial reference for the service would result in the new URL:

corbaloc:555objs.com/NotificationService

The implementation of the ORB:resolve_initial_references operation would take the newly constructed URL and call CORBA::ORB::string_to_object to obtain the initial reference for the service.

The URL specified as the value of the -ORBDefaultInitRef argument can contain more than a single location. This is similar to the functionality provided for the list of locations to be used by the Tobj_Bootstrap object. In this situation, the ORB will process the locations in the URL based on the syntax rules for the URL. For example, if the following was specified as the default initial reference argument:

-ORBDefaultInitRef corbaloc:555objs.com,555Backup.com

A call to ORB:resolve_initial_references("NameService") to obtain the initial reference for the service would result in one of the following new URLs:

corbaloc:555objs.com/NameService

or:

corbaloc:555Backup.com/NameService

The resulting URL would then be passed to CORBA::ORB::string_to_object in order to obtain the initial reference for the service.
When establishing a network link, this is the minimum level of encryption required. Zero (0) means no encryption, while 40, 56, and 128 specify the length (in bits) of the encryption key. If this minimum level of encryption cannot be met, link establishment will fail.

The default is 0.

When establishing a network link, this is the maximum level of encryption allowed. Zero (0) means no encryption, while 40, 56, and 128 specify the length (in bits) of the encryption key. The default is whatever capability is specified by the license. The –ORBmaxCrypto or –ORBmaxCrypto options are available only if either the International or U.S. Canada BEA Tuxedo Security Add-on Package is installed.

Specifies that certificate-based authentication should be enabled when accepting an SSL connection from a remote application.

The –ORBmutualAuth option is available only if either the International or U.S. Canada BEA Tuxedo Security Add-on Package is installed.

Determines how the BEA Tuxedo CORBA ORB will behave when a digital certificate for a peer of an outbound connection initiated by the BEA Tuxedo ORB is received as part of the Secure Socket Layer (SSL) protocol handshake. The validation is only performed by the initiator of a secure connection and confirms that the peer server is actually located at the same network address specified by the domain name in the server’s digital certificate. This validation is not technically part of the SSL protocol, but is similar to the same check done in web browsers.

A value of detect causes an BEA Tuxedo CORBA ORB to verify that the host specified in the object reference used to make the connection matches the domain name specified in the peer’s digital certificate. If the comparison fails, the ORB refuses to authenticate the peer and drops the connection. This check protects against man-in-the-middle attacks.

A value of warn causes a BEA Tuxedo CORBA ORB to verify that the host specified in the object reference used to make the connection matches the domain name specified in the peer’s digital certificate. If the comparison fails, the ORB logs a message to the user log, but continues processing the connection.

A value of none causes a BEA Tuxedo CORBA ORB not to perform the peer validation and will continue the processing of the connection.

The –ORBpeerValidate option is available only if either the International or U.S. Canada BEA Tuxedo Security Add-on Package is installed.

If not specified, the default is detect.
[–ORBport port-number]
Specifies the network address to be used by the ORB to accept connections from remote CORBA clients. Typically, a system administrator assigns the port number for the client from the "user" range of port numbers rather from the dynamic range. This keeps the joint client/servers from using conflicting ports. This parameter is required in order for the BEA Tuxedo CORBA ORB to create persistent object references. Persistent objects references must be served on the same port after that is contained in the object reference, even if the ORB has been restarted. For transient object references, any port is sufficient and can be obtained by the ORB dynamically. The port-number is the TCP port number at which the BEA Tuxedo CORBA ORB process listens for incoming requests. The port-number can be a number between 0 and 65535.

[–ORBsecurePort port-number]
Specifies the port number that the IIOP Listener/Handler should use to listen for secure connections using the Secure Socket Layer protocol. If the command-line option is specified without a port number, then the OMG assigned port number 684 will be used for SSL connections. The port-number is the TCP port number at which the BEA Tuxedo CORBA ORB process listens for incoming requests. The port-number can be a number between 0 and 65535.

An administrator can configure to only allow secure connections into the BEA Tuxedo CORBA ORB by setting port numbers specified by the –ORBport and –ORBsecurePort to the same value. The –ORBsecurePort option is available only if either the International or U.S_Canada BEA Tuxedo Security Add-on Package is installed.

Portability
The BEA Tuxedo CORBA ORB is supported as a BEA Tuxedo-supplied client or server on UNIX and Microsoft Windows operating systems. It is also supported as a BEA Tuxedo-supplied client on the Windows XP operating systems.

Interoperability
The BEA Tuxedo CORBA ORB will interoperate with any IIOP compliant ORB that supports the 1.0, 1.1, or 1.2 version of the GIOP protocol over a TCP/IP connection. In addition, the BEA Tuxedo CORBA ORB will interoperate with any IIOP-SSL compliant ORB that supports the use of the TAG_SSL_SEC_TRANS tagged component in object references and version 3 of the Secure Socket Layer protocol.
Examples

**C++ code example**

```cpp
ChatClient -ORBId BEA_IIoP -ORBport 2100
-ORBDefaultInitRef corbaloc:piglet:1900
-ORBInitRef TraderService=corbaloc:owl:2530
-ORBsecurePort 2100
-ORBminCrypto 40
-ORBmaxCrypto 128
TechTopics
```

**Java code example**

```java
java -DORBDefaultInitRef=corbalocs:piglet:1900
.....-DORBInitRef=TraderService=corbaloc:owl:2530
-Dorg.omg.CORBA.ORBPort=1948
-classpath=%CLASSPATH% client
```

See Also

ISL

**Policy Member Functions**

A policy is an object used to communicate certain choices to an ORB regarding its operation. This information is accessed in a structured manner using interfaces derived from the Policy interface defined in the CORBA module.

**Note:** These CORBA::Policy operations and structures are not usually needed by programmers. The derived interfaces usually contain the information relevant to specifications. A policy object can be constructed by a specific factory or by using the CORBA::create_policy operation.

The mapping of this object to C++ is as follows:

```cpp
class CORBA
{
    class Policy
    {
        public:
            copy();
            void destroy();
```
PolicyList is used the same as any other C++ sequence mapping. For a discussion of sequence usage, see Sequences.

See Also:
POA Policy and CORBA::ORB::create_policy.

**CORBA::Policy::copy**

**Synopsis**
Copies the policy object.

**C++ Binding**
```cpp
CORBA::Policy::copy();
```

**Argument**
None.

**Description**
This operation copies the policy object. The copy does not retain any relationships that the policy had with any domain or object.

**Note:** This function is supported only for a joint client/server.

**Return Values**
None.

**CORBA::Policy::destroy**

**Synopsis**
Destroys the policy object.

**C++ Binding**
```cpp
void CORBA::Policy::destroy();
```
Argument
None.

Exceptions
If the policy object determines that it cannot be destroyed, the CORBA::NO_PERMISSION exception is raised.

Description
This operation destroys the policy object. It is the responsibility of the policy object to determine whether it can be destroyed.

Note: This function is supported only for a joint client/server.

Return Values
None.

PortableServer Member Functions
The mapping of the PortableServer member functions to C++ is as follows:

```cpp
// C++
class PortableServer
{
    public:
        class LifespanPolicy;
        class IdAssignmentPolicy;
        class POA::find_POA
        class reference_to_id
        class POAManager;
        class POA;
        class Current;
        class virtual ObjectId
        class ServantBase
};
```

ObjectId
A value that is used by the POA and by the user-supplied implementation to identify a particular abstract CORBA object. ObjectId values may be assigned and managed by the POA, or they may be assigned and managed by the implementation. ObjectId values are
hidden from clients, encapsulated by references. ObjectIds have no standard form; they are managed by the POA as uninterpreted octet sequences.

The following sections describe the remaining classes.

**PortableServer::POA::activate_object**

**Synopsis**

Explicitly activates an individual object.

**C++ Binding**

```cpp
ObjectId * activate_object (
   Servant p_servant);
```

**Argument**

- `p_servant`
  - An instance of the C++ implementation class for the interface.

**Exceptions**

- If the specified servant is already in the Active Object Map, the ServantAlreadyActive exception is raised.

  **Note:** Other exceptions can occur if the POA uses unsupported policies.

**Description**

This operation explicitly activates an individual object by generating an ObjectId and entering the ObjectId and the specified servant in the Active Object Map.

**Note:** This function is supported only for a joint client/server.

**Return Values**

- If the function succeeds, the ObjectId is returned.

**Example**

In the following example, the first struct creates a servant by a user-defined constructor. The second struct tells the POA that the servant can be used to handle requests on an object. The POA returns the ObjectId it has created for the object. The third statement assumes that the POA has the IMPLICIT_ACTIVATION policy (the only supported policy in version 4.2 of the BEA Tuxedo software) and returns a reference to the object. That reference can then be handed to a client for
invocations. When the client invokes on the reference, the request is returned to the servant just created.

MyFooServant* afoo = new MyFooServant(poa,27);
PortableServer::Object_id_var oid =
    poa->activate_object(afoo);
Foo_var foo = afoo->_this();

PortableServer::POA::activate_object_with_id

Synopsis
Activates an individual object with a specified ObjectId.

C++ Binding
void activate_object_with_id (  
    const ObjectId & id,  
    Servant p_servant);

Argument
id
    ObjectId that identifies the object on which that operation was invoked.

p_servant
    An instance of the C++ implementation class for the interface.

Exceptions
The ObjectAlreadyActive exception is raised if the CORBA object denoted by the ObjectId value is already active in this POA.

The ServantAlreadyActive exception is raised if the servant is already in the Active Object Map.

Note: Other exceptions can occur if the POA uses unsupported policies.

The BAD_PARAM system exception may be raised if the POA has the SYSTEM_ID policy and it detects that the ObjectId value was not generated by the system or for this POA. An ORB is not required to detect all such invalid ObjectId values. However, a portable application must not invoke activate_object_with_id on a POA if the POA has the SYSTEM_ID policy with an ObjectId value that was not previously generated by the system for that POA, or, if the POA also has the PERSISTENT policy, for a previous instantiation of the same POA.
PortableServer Member Functions

Description
This operation enters an association between the specified ObjectId and the specified servant in the Active Object Map.

Note: This function is supported only for a joint client/server.

Return Values
None.

Example
MyFooServant* afoo = new MyFooServant(poa, 27);
PortableServer::ObjectId_var oid =
    PortableServer::string_to_ObjectId("myLittleFoo");
poa->activate_object_with_id(oid.in(), afoo);
Foo_var foo = afoo->_this();

PortableServer::POA::create_id_assignment_policy

Synopsis
Obtains an object with the IdAssignmentPolicy interface so the user can pass the object to the POA::create_POA operation.

C++ Binding
IdAssignmentPolicy_ptr
PortableServer::POA::create_id_assignment_policy (PortableServer::IdAssignmentPolicyValue value)

Argument
value
A value of either PortableServer::USER_ID, indicating ObjectIds are assigned only by the application, or PortableServer::SYSTEM_ID, indicating ObjectIds are assigned only by the system.

Description
The POA::create_id_assignment_policy operation obtains objects with the IdAssignmentPolicy interface. When passed to the POA::create_POA operation, this policy specifies whether ObjectIds in the created POA are generated by the application or by the ORB. The following values can be supplied:
PortableServer::USER_ID—objects created with that POA are assigned ObjectIds only by the application.

PortableServer::SYSTEM_ID—objects created with that POA are assigned ObjectIds only by the POA. If the POA also has the PERSISTENT LifespanPolicy, assigned ObjectIds must be unique across all instantiations of the same POA.

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

Note: This function is supported only for a joint client/server.

Return Values

Returns an Id Assignment policy.

PortableServer::POA::create_lifespan_policy

Synopsis

Obtains an object with the LifespanPolicy interface so the user can pass the object to the POA::create_POA operation.

C++ Binding

LifespanPolicy_ptr

PortableServer::POA::create_lifespan_policy (PortableServer::LifespanPolicyPolicyValue value)

Argument

value

A value of either PortableServer::USER_ID, indicating ObjectIds are assigned only by the application, or PortableServer::SYSTEM_ID, indicating ObjectIds are assigned only by the system.

Description

Objects with the LifespanPolicy interface are obtained using the POA::create_lifespan_policy operation and passed to the POA::create_POA operation to specify the lifespan of the objects implemented in the created POA. The following values can be supplied.

- TRANSIENT—the objects implemented in the POA cannot outlive the process in which they are first created. Once the POA is deactivated, use of any object references generated from it will result in an OBJECT_NOT_EXIST exception.
**PERSISTENT**—the objects implemented in the POA can outlive the process in which they are first created.

- Persistent objects have a POA associated with them (the POA which created them). When the ORB receives a request on a persistent object, it first searches for the matching POA, based on the names of the POA and all of its ancestors.
- Administrative action beyond the scope of this specification may be necessary to inform the ORB's location service of the creation and eventual termination of existence of this POA, and optionally to arrange for on-demand activation of a process implementing this POA.
- POA names must be unique within their enclosing scope (the parent POA). A portable program can assume that POA names used in other processes will not conflict with its own POA names. A conforming CORBA implementation will provide a method for ensuring this property.

If no **LifespanPolicy** object is passed to **POA::create_POA**, the lifespan policy defaults to **TRANSIENT**.

**Note:** This function is supported only for a joint client/server.

**Return Values**

Returns a **LifespanPolicy**.

**PortableServer::POA::create_POA**

**Synopsis**

Creates a new POA as a child of the target POA.

**C++ Binding**

```cpp
POA_ptr PortableServer::create_POA (  
    const char * adapter_name,  
    POAManager_ptr a_POAManager,  
    const CORBA::PolicyList & policies)
```

**Arguments**

- **adapter_name**
  
  The name of the POA to be created.
a_POAManager
    Either a NULL value, indicating that a new POAManager is to be created and associated
    with the new POA, or a pointer to an existing POAManager.

policies
    Policy objects to be associated with the new POA.

Exceptions

AdapterAlreadyExists
    Raised if the target POA already has a child POA with the specified name.

InvalidPolicy
    Raised if any of the policy objects specified are not valid for the ORB implementation, if
    conflicting policy objects are specified, or if any of the specified policy objects require
    prior administrative action that has not been performed. This exception contains the index
    in the policy parameter value of the first offending policy object.

IMP_LIMIT
    Raised if the program tries to create a POA with a LifespanPolicy of PERSISTENT without
    having set a port, as described in the operation CORBA::ORB_init.

Description

This operation creates a new POA as a child of the target POA. The specified name, which must
be unique, identifies the new POA with respect to other POAs with the same parent POA.

If the a_POAManager parameter is NULL, a new PortableServer::POAManager object is
created and associated with the new POA. Otherwise, the specified POAManager object is
associated with the new POA. The POAManager object can be obtained using the attribute name
the_POAManager.

The specified policy objects are associated with the POA and are used to control its behavior. The
policy objects are effectively copied before this operation returns, so the application is free to
destroy them while the POA is in use. Policies are not inherited from the parent POA.

Note: This function is supported only for joint client/servers.

Return Values

Returns a pointer to the POA that was created.

Examples

Example 1
In this example, the child POA would use the same manager as the parent POA; the child POA would then have the same state as the parent (that is, it would be active if the parent is active).

```c++
CORBA::PolicyList policies(2);
policies.length (1);
policies[0] = rootPOA->create_lifespan_policy(
    PortableServer::LifespanPolicy::TRANSIENT);
PortableServer::POA_ptr poa =
    rootPOA->create_POA("my_little_poa",
    rootPOA->the_POAManager, policies);
```

**Example 2**

In this example, a new POA is created as a child of the root POA.

```c++
CORBA::PolicyList policies(2);
policies.length (1);
policies[0] = rootPOA->create_lifespan_policy(
    PortableServer::LifespanPolicy::TRANSIENT);
PortableServer::POA_ptr poa =
    rootPOA->create_POA("my_little_poa",
    PortableServer::POAManager::_nil(), policies);
```

**PortableServer::POA::create_reference**

**Synopsis**

Creates an object reference that encapsulates a POA-generated `ObjectId` value and the specified interface repository ID.

**C++ Binding**

```c++
CORBA::Object_ptr create_reference (const char * intf)
```

**Argument**

- `intf`
  
  The interface repository ID.

**Exceptions**

This operation requires the LifespanPolicy to have the value `SYSTEM_ID`; if not present, the `PortableServer::WrongPolicy` exception is raised.
Description

This `create_reference` operation creates an object reference that encapsulates a POA-generated `ObjectId` value and the specified interface repository ID. This operation collects the necessary information to constitute the reference from information associated with the POA and from parameters to the operation. This operation only creates a reference; it does not associate the reference with an active servant. The resulting reference may be passed to clients, so that subsequent requests on those references return to the POA using the `ObjectId` generated. The generated `ObjectId` value may be obtained by invoking `POA::reference_to_id` with the created reference.

**Note:** This function is supported only for a joint client/server.

Return Values

Returns a pointer to the object.

**PortableServer::POA::create_reference_with_id**

Synopsis

Creates an object reference that encapsulates the specified `ObjectId` and interface repository ID values.

C++ Binding

```
CORBA::Object_ptr create_reference_with_id (  
    const ObjectId & oid,  
    const char * intf)
```

Arguments

- `oid`  
  `ObjectId` that identifies the object on which that operation was invoked.

- `intf`  
  The interface repository ID.

Exceptions

If the POA has a LifespanPolicy with value `SYSTEM_ID` and it detects that the `ObjectId` value was not generated by the system or for this POA, the operation will raise the `BAD_PARAM` system exception.
**Description**

The `create_reference` operation creates an object reference that encapsulates the specified `ObjectId` and interface repository ID values. This operation collects the necessary information to constitute the reference from information associated with the POA and from parameters to the operation. This operation only creates a reference; it does not associate the reference with an active servant. The resulting reference may be passed to clients, so that subsequent requests on those references cause the invocation to be returned to the same POA with `ObjectId` specified.

**Note:** This function is supported only for a joint client/server.

**Return Values**

Returns `Object_ptr`.

**Example**

```cpp
PortableServer::ObjectId_var oid = PortableServer::string_to_ObjectId("myLittleFoo");
CORBA::Object_var obj = poa->create_reference_with_id(
    oid.in(), "IDL:Foo:1.0");
Foo_var foo = Foo::_narrow(obj);
```

**PortableServer::POA::deactivate_object**

**Synopsis**

Removes the `ObjectId` from the Active Object Map.

**C++ Binding**

```cpp
void deactivate_object (
    const ObjectId & oid)
```

**Argument**

`oid`

`ObjectId` that identifies the object.

**Exceptions**

If there is no active object associated with the specified `ObjectId`, the operation raises an `ObjectNotActive` exception.
Description

This operation causes the association of the ObjectId specified by the oid parameter and its servant to be removed from the Active Object Map.

Note: This function is supported only for a joint client/server.

Return Values

None.

**PortableServer::POA::destroy**

Synopsis

Destroys the POA and all descendant POAs.

C++ Binding

```cpp
void destroy (
   CORBA::Boolean etherealize_objects,
   CORBA::Boolean wait_for_completion)
```

Arguments

- **etherealize_objects**
  
  This argument should be **FALSE** for this release of BEA Tuxedo.

- **wait_for_completion**
  
  This argument indicates whether or not the operation should return immediately.

Description

This operation destroys the POA and all descendant POAs. The POA with its name may be recreated later in the same process. (This differs from the **POAManager::deactivate** operation, which does not allow a recreation of its associated POA in the same process.)

When a POA is destroyed, any requests that have started execution continue to completion. Any requests that have not started execution are processed as if they were newly arrived and there is no POA; that is, they are rejected and the **OBJECT_NON_EXIST** exception is raised.

If the **wait_for_completion** parameter is **TRUE**, the **destroy** operation returns only after all requests in process have completed and all invocations of **etherealize** have completed. Otherwise, the **destroy** operation returns after destroying the POAs.
Note: This release of BEA Tuxedo does not support multithreading. Hence, `wait_for_completion` should not be `TRUE` if the call is made in the context of an object invocation. That is, the POA cannot start destroying itself if it is currently executing.

Note: This function is supported only for a joint client/server.

Return Values
None.

**PortableServer::POA::find_POA**

Synopsis
Returns a reference to a child POA with a given name.

C++ Binding

```cpp
void find_POA( in string adapter_name, in boolean activate_it);
```

Argument

- **adapter_name**
  A reference to the target POA.

- **activate_it**
  In this version of BEA Tuxedo, this parameter must be `FALSE`.

Exception

- **AdapterNonExistent**
  This exception is raised if the POA does not exist.

Description

If the POA has a child POA with the specified name, that child POA is returned. If a child POA with the specified name does not exist and the value of the `activate_it` parameter is `FALSE`, the `AdapterNonExistent` exception is raised.

Return Values
None.
**PortableServer::POA::reference_to_id**

**Synopsis**

Returns the ObjectId value encapsulated by the specified reference.

**C++ Binding**

```cpp
ObjectId reference_to_id(in Object reference);
```

**Argument**

`reference`

Specifies the reference to the object.

**Exceptions**

`WrongAdapter`

This exception is raised if the reference was not created by that POA.

**Description**

This operation returns the ObjectId value encapsulated by the specified reference. This operation is valid only if the reference was created by the POA on which the operation is being performed. The object denoted by the reference does not have to be active for this operation to succeed.

**Note:** This function is supported only for a joint client/server.

**Return Values**

Returns the ObjectId value encapsulated by the specified reference.

---

**PortableServer::POA::the_POAManager**

**Synopsis**

Identifies the POA manager associated with the POA.

**C++ Binding**

```cpp
POAManager_ptr the_POAManager ();
```
PortableServer Member Functions

Argument

None.

Description
This read-only attribute identifies the POA manager associated with the POA.

Note: This function is supported only for a joint client/server.

Return Values
None.

Example
poa->the_POAManager()->activate();

This statement will set the state of the POAManager for the given POA to active, which is required if the POA is to accept requests. Note that if the POA has a parent, that is, it is not the root POA, all of its parent’s POAManagers must also be in the active state for this statement to have any effect.

PortableServer::ServantBase::_default_POA

Synopsis
Returns an object reference to the POA associated with the servant.

C++ Binding

```cpp
class PortableServer
{
    class ServantBase
    {
        public:
            virtual POA_ptr _default_POA();
    }
}
```
Argument

None.

Description

All C++ Servants inherit from PortableServer::ServantBase, so they all inherit the _default_POA function. In this version of BEA Tuxedo there is usually no reason to use _default_POA.

The default implementation of this function returns an object reference to the root POA of the default ORB in this process—the same as the return value of an invocation of ORB::resolve_initial_references("RootPOA"). A C++ servant can override this definition to return the POA of its choice, if desired.

Note: This function is supported only for joint client/servers.

Return Values

The default POA associated with the servant.

POA Current Member Functions

The PortableServer::Current interface, derived from CORBA::Current, provides method implementations with access to the identity of the object on which the method was invoked.

PortableServer::Current::get_object_id

Synopsis

Returns the ObjectId identifying the object in whose context it is called.

C++ Binding

ObjectId * get_object_id ()

Arguments

None.

Exception

If called outside the context of a POA-dispatched operation, a PortableServer::NoContext exception is raised.
Description

This operation returns the PortableServer::ObjectId identifying the object in whose context it is called.

Note: This function is supported only for a joint client/server.

Return Values

This operation returns the ObjectId identifying the object in whose context it is called.

PortableServer::Current::get_POA

Synopsis

Returns a reference to the POA implementing the object in whose context it is called.

C++ Binding

POA_ptr get_POA ();

Argument

None.

Exceptions

If this operation is called outside the context of a POA-dispatched operation, a PortableServer::NoContext exception is raised.

Description

This operation returns a reference to the POA implementing the object in whose context it is called.

Note: This function is supported only for a joint client/server.

Return Values

This operation returns a reference to the POA implementing the object in whose context it is called.

POAManager Member Functions

Each POA object has an associated POAManager object. A POAManager may be associated with one or more POA objects. A POAManager encapsulates the processing state of the POAs with
which it is associated. Using operations on the POA manager, an application can cause requests for those POAs to be queued or discarded, and can cause the POAs to be deactivated.

POA managers are created and destroyed implicitly. Unless an explicit POAManager object is provided at POA creation time, a POAManager is created when a POA is created and is automatically associated with that POA. A POAManager object is implicitly destroyed when all of its associated POAs have been destroyed.

A POAManager has four possible processing states: active, inactive, holding, and discarding. The processing state determines the capabilities of the associated POAs and the disposition of requests received by those POAs.

A POAManager is created in the holding state. In that state, any invocations on its POA are queued until the POA manager enters the active state. This version of BEA Tuxedo supports only the ability to enter active and inactive states. That is, this version does not support the ability to return to holding state or to enter discarding state.

**PortableServer::POAManager::activate**

**Synopsis**

Changes the state of the POAManager to active.

**C++ Binding**

```cpp
void activate();
```

**Argument**

None.

**Exceptions**

If this operation is issued while the POAManager is in the inactive state, the PortableServer::POAManager::AdapterInactive exception is raised.

**Description**

This operation changes the state of the POAManager to active. Entering the active state enables the associated POAs to process requests.

**Note:** All parent POAs must also have POA Managers in the active state for this POA to process requests.

**Note:** This function is supported only for a joint client/server.
Return Values
None.

**PortableServer::POAManager::deactivate**

**Synopsis**
Changes the state of the POA manager to *inactive*.

**C++ Binding**

```cpp
void deactivate (
    CORBA::Boolean etherealize_objects,
    CORBA::Boolean wait_for_completion);
```

**Argument**

- *etherealize_objects*
  For BEA WebLogic Enterprise 4.2 software and later software and BEA Tuxedo 8.0 and later software, this argument should always be set to `FALSE`.

- *wait_for_completion*
  If this argument is `TRUE`, the `deactivate` operation returns only after all requests in process have completed. If this argument is `FALSE`, the `deactivate` operation returns after changing the state of the associated POAs.

**Exceptions**

If issued while the POA manager is in the *inactive* state, the `PortableServer::POAManager::AdapterInactive` exception is raised.

**Description**
This operation changes the state of the POAManager to *inactive*. Entering the inactive state causes the associated POAs to reject requests that have not begun to be executed, as well as any new requests.

**Note:** This release of BEA Tuxedo does not support multithreading. Hence, `wait_for_completion` should not be `TRUE` if the call is made in the context of an object invocation. That is, the POAManager cannot be set to inactive state if it is currently executing.

**Note:** This function is supported only for a joint client/server.
Return Values
None.

POA Policy Member Objects
Interfaces derived from CORBA::Policy are used with the POA::create_POA operation to specify policies that apply to a POA. Policy objects are created using factory operations on any preexisting POA, such as the root POA. Policy objects are specified when a POA is created. Policies may not be changed on an existing POA. Policies are not inherited from the parent POA.

PortableServer::LifespanPolicy

Synopsis
Specifies the life span of objects to the create_POA operation.

Description
Objects with the LifespanPolicy interface are obtained using the POA::create_lifespan_policy operation and are passed to the POA::create_POA operation to specify the life span of the objects implemented in the created POA. The following values can be supplied:

- **TRANSIENT**—the objects implemented in the POA cannot outlive the process in which they are first created.
- **PERSISTENT**—the objects implemented in the POA can outlive the process in which they are first created.

Persistent objects have a POA associated with them (the POA that created them). When the ORB receives a request on a persistent object, it searches for the matching POA, based on the names of the POA and all of its ancestors.

POA names must be unique within their enclosing scope (the parent POA). A portable program can assume that POA names used in other processes will not conflict with its own POA names.

If no LifespanPolicy object is passed to create_POA, the lifespan policy defaults to TRANSIENT.

**Note:** This function is supported only for a joint client/server.
Exceptions

None.

**PortableServer::IdAssignmentPolicy**

**Synopsis**

Specifies whether ObjectIds in the created POA are generated by the application or by the ORB.

**Description**

Objects with the IdAssignmentPolicy interface are obtained using the POA::create_id_assignment_policy operation and are passed to the POA::create_POA operation to specify whether ObjectIds in the created POA are generated by the application or by the ORB. The following values can be supplied:

- **USER_ID**—objects created with that POA are assigned ObjectIds only by the application.
- **SYSTEM_ID**—objects created with that POA are assigned ObjectIds only by the POA. If the POA also has the PERSISTENT policy, assigned ObjectIds must be unique across all instantiations of the same POA.

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

**Note:** This function is supported only for a joint client/server.

**Request Member Functions**

The mapping of these member functions to C++ is as follows:

```cpp
// C++
class Request
{
public:
    Object_ptr target() const;
    const char *operation() const;
    NamedValue_ptr result();
    NVList_ptr arguments();
    Environment_ptr env();
    ExceptionList_ptr exceptions();
    ContextList_ptr contexts();
};
```
void ctx(Context_ptr);
Context_ptr ctx() const

// argument manipulation helper functions
Any &add_in_arg();
Any &add_in_arg(const char* name);
Any &add_inout_arg();
Any &add_inout_arg(const char* name);
Any &add_out_arg();
Any &add_out_arg(const char* name);
void set_return_type(TypeCode_ptr tc);
Any &return_value();

void invoke();
void send_oneway();
void send_deferred();
void get_response();
Boolean poll_response();

Note: The add_*_arg, set_return_type, and return_value member functions are added as shortcuts for using the attribute-based accessors.
The following sections describe these member functions.

**CORBA::Request::arguments**

**Synopsis**
Retrieves the argument list for the request.

**C++ Binding**
CORBA::NVList_ptr CORBA::Request::arguments () const;

**Arguments**
None.

**Description**
This member function retrieves the argument list for the request. The arguments can be input, output, or both.
Return Values

If the function succeeds, the value returned is a pointer to the list of arguments to the operation for the request. The returned argument list is owned by the Request object reference and should not be released.

If the function does not succeed, an exception is thrown.

**CORBA::Request::ctx(Context_ptr)**

Synopsis

Sets the Context object for the operation.

C++ Binding

```cpp
void CORBA::Request::ctx (  
    CORBA::Context_ptr CtxObject);
```

Argument

`CtxObject`

The new value to which to set the Context object.

Description

This member function sets the Context object for the operation.

Return Values

None.

See Also

`CORBA::Request::ctx()`

**CORBA::Request::get_response**

Synopsis

Retrieves the response of a specific deferred synchronous request.

C++ Binding

```cpp
void CORBA::Request::get_response ();
```
Arguments
None.

Description
This member function retrieves the response of a specific request; it is used after a call to the
CORBA::Request::send_deferred function or the
CORBA::Request::send_multiple_requests function. If the request has not completed, the
CORBA::Request::get_response function blocks until it does complete.

Return Values
None.

See Also
CORBA::Request::send_deferred

CORBA::Request::invoke

Synopsis
Performs an invoke on the operation specified in the request.

C++ Binding

    void CORBA::Request::invoke ();

Arguments
None.

Description
This member function calls the Object Request Broker (ORB) to send the request to the
appropriate server application.

Return Values
None.
**CORBA::Request::operation**

**Synopsis**
Retrieves the operation intended for the request.

**C++ Binding**
```
const char * CORBA::Request::operation () const;
```

**Arguments**
None.

**Description**
This member function retrieves the operation intended for the request.

**Return Values**
If the function succeeds, the value returned is a pointer to the operation intended for the object; the value can be 0 (zero). The memory returned is owned by the Request object and should not be freed.

If the function does not succeed, an exception is thrown.

**CORBA::Request::poll_response**

**Synopsis**
Determines whether a deferred synchronous request has completed.

**C++ Binding**
```
CORBA::Boolean CORBA::Request::poll_response ();
```

**Arguments**
None.

**Description**
This member function determines whether the request has completed and returns immediately. You can use this call to check the state of the request. This member function can also be used to determine whether a call to `CORBA::Request::get_response` will block.
Return Values

If the function succeeds, the value returned is `CORBA_TRUE` if the response has already completed, and `CORBA_FALSE` if the response has not yet completed.

If the function does not succeed, an exception is thrown.

See Also

- `CORBA::ORB::get_next_response`
- `CORBA::ORB::poll_next_response`
- `CORBA::ORB::send_multiple_requests`
- `CORBA::Request::get_response`
- `CORBA::Request::send_deferred`

**CORBA::Request::result**

Synopsis

Retrieves the result of the request.

C++ Binding

```cpp
CORBA::NamedValue_ptr CORBA::Request::result ();
```

Arguments

None.

Description

This member function retrieves the result of the request.

Return Values

If the function succeeds, the value returned is a pointer to the result of the operation. The returned result is owned by the Request object and should not be released.

If the function does not succeed, an exception is thrown.

**CORBA::Request::env**

Synopsis

Retrieves the environment of the request.
Request Member Functions

C++ Binding

```cpp
CORBA::Environment_ptr CORBA::Request::env ();
```

Arguments

None.

Description

This member function retrieves the environment of the request.

Return Values

- If the function succeeds, the value returned is a pointer to the environment of the operation. The returned environment is owned by the Request object and should not be released.
- If the function does not succeed, an exception is thrown.

**CORBA::Request::ctx**

Synopsis

Retrieves the context of the request.

C++ Binding

```cpp
CORBA::context_ptr CORBA::Request::ctx ();
```

Arguments

None.

Description

This member function retrieves the context of the request.

Return Values

- If the function succeeds, the value returned is a pointer to the context of the operation. The returned context is owned by the Request object and should not be released.
- If the function does not succeed, an exception is thrown.
**CORBA::Request::contexts**

**Synopsis**  
Retrieves the context lists for the request.

**C++ Binding**  
```cpp
CORBA::ContextList_ptr CORBA::Request::contexts ();
```

**Arguments**  
None.

**Description**  
This member function retrieves the context lists for the request.

**Return Values**  
If the function succeeds, the value returned is a pointer to the context lists for the operation. The returned context list is owned by the Request object and should not be released.

If the function does not succeed, an exception is thrown.

**CORBA::Request::exceptions**

**Synopsis**  
Retrieves the exception lists for the request.

**C++ Binding**  
```cpp
CORBA::ExceptionList_ptr CORBA::Request::exceptions ();
```

**Arguments**  
None.

**Description**  
This member function retrieves the exception lists for the request.

**Return Values**  
If the function succeeds, the value returned is a pointer to the exception list for the request. The returned exception list is owned by the Request object and should not be released.
If the function does not succeed, an exception is thrown.

**CORBA::Request::target**

Synopsis
Retrieves the target object reference for the request.

C++ Binding
```
CORBA::Object_ptr CORBA::Request::target () const;
```

Arguments
None.

Description
This member function retrieves the target object reference for the request.

Return Values
- If the function succeeds, the value returned is a pointer to the target object of the operation. The returned value is owned by the Request object and should not be released.
- If the function does not succeed, an exception is thrown.

**CORBA::Request::send_deferred**

Synopsis
Initiates a deferred synchronous request.

C++ Binding
```
void CORBA::Request::send_deferred ();
```

Arguments
None.

Description
This member function initiates a deferred synchronous request. You use this function when a response is expected and in conjunction with the `CORBA::Request::get_response` function.
Return Values

None.

See Also

CORBA::ORB::get_next_response
CORBA::ORB::poll_next_response
CORBA::ORB::send_multiple_requests
CORBA::Request::get_response
CORBA::Request::poll_response
CORBA::Request::send_oneway

**CORBA::Request::send_oneway**

Synopsis

Initiates a one-way request.

C++ Binding

```cpp
void CORBA::Request::send_oneway ();
```

Arguments

None.

Description

This member function initiates a one-way request; it does not expect a response.

Return Values

None.

See Also

CORBA::ORB::send_multiple_requests
CORBA::Request::send_deferred

Strings

The mapping of these functions to C++ is as follows:

```cpp
// C++
```
namespace CORBA {
    static char * string_alloc(ULong len);
    static char * string_dup (const char *);
    static void string_free(char *);
    ...
}

Note: A static array of char in C++ decays to a char*. Therefore, care must be taken when
assigning a static array to a String_var, because the String_var assumes that the
pointer points to data allocated via string_alloc, and thus eventually attempts to free
it using string_free.

This behavior has changed in ANSI/ISO C++, where string literals are const char*, not
char*. However, since most C++ compilers do not yet implement this change, portable
programs must heed the advice given here.

The following sections describe the functions that manage memory allocated to strings.

**CORBA::string_alloc**

**Synopsis**
Allocates memory for a string.

**C++ Binding**

```cpp
cpp
char * CORBA::string_alloc(ULong len);
```

**Argument**

len

The length of the string for which to allocate memory.

**Description**

This member function dynamically allocates memory for a string, or returns a nil pointer if it
cannot perform the allocation. It allocates len+1 characters so that the resulting string has
enough space to hold a trailing NULL character. Free the memory allocated by this member
function by calling the CORBA::string_free member function.

This function does not throw CORBA exceptions.
Return Values

If the function succeeds, the return value is a pointer to the newly allocated memory for the string object; if the function fails, the return value is a nil pointer.

Example

cchar* s = CORBA::string_alloc(10);

See Also

CORBA::string_free
CORBA::string_dup

CORBA::string_dup

Synopsis

Makes a copy of a string.

C++ Binding

char * CORBA::string_dup (const char * Str);

Argument

Str

The address of the string to be copied.

Description

This function dynamically allocates enough memory to hold a copy of its string argument, including the NULL character, copies the string argument into that memory, and returns a pointer to the new string.

This function does not throw CORBA exceptions.

Return Values

If the function succeeds, the return value is a pointer to the new string; if the function fails, the return value is a nil pointer.

Example

cchar* s = CORBA::string_dup("hello world");
Strings

See Also

CORBA::string_free
CORBA::string_alloc

**CORBA::string_free**

**Synopsis**

Frees memory allocated to a string.

**C++ Binding**

```cpp
void CORBA::string_free(char * Str);
```

**Argument**

*Str*

The address of the memory to be deallocated.

**Description**

This member function deallocates memory that was previously allocated to a string using the CORBA::string_alloc() or CORBA::string_dup() member functions. Passing a nil pointer to this function is acceptable and results in no action being performed.

This function may not throw CORBA exceptions.

**Return Values**

None.

**Example**

```cpp
char* s = CORBA::string_dup("hello world");
CORBA::string_free(s);
```

See Also

CORBA::string_alloc
CORBA::string_dup
Wide Strings

Both bounded and unbounded wide string types are mapped to CORBA::WChar* in C++. In addition, the CORBA module defines WString_var and WString_out classes. Each of these classes provides the same member functions with the same semantics as their string counterparts, except of course they deal with wide strings and wide characters.

Dynamic allocation and deallocation of wide strings must be performed via the following functions:

```c++
namespace CORBA {
    // ...
    WChar *wstring_alloc(ULong len);
    WChar *wstring_dup(const WChar* ws);
    void wstring_free(WChar*);
}
```

These member functions have the same semantics as the same functions for the string type, except they operate on wide strings.

A compliant mapping implementation provides overloaded `operator<<` (insertion) and `operator>>` (extraction) operators for using WString_var and WString_out directly with C++ iostreams.

For descriptions of these member functions, see the corresponding function for Strings.

Listing 14-1 shows a code example that uses wide strings and wide characters.

**Listing 14-1  Wide Strings Example**

```c++
// Get a string from the user:
cout << "String?";
char mixed[256]; // this should be big enough!
char lower[256];
char upper[256];
wchar_t wmixed[256];
cin >> mixed;
// Convert the string to a wide char string,
```
// because this is what the server will expect.
mbstowcs(wmixed, mixed, 256);

// Convert the string to upper case:
CORBA::WString_var v_upper = CORBA::wstring_dup(wmixed);
v_simple->to_upper(v_upper.inout());
wcstombs(upper, v_upper.in(), 256);
cout << upper << endl;

// Convert the string to lower case:
CORBA::WString_var v_lower = v_simple->to_lower(wmixed);
wcestombs(lower, v_lower.in(), 256);
cout << lower << endl;

// Everything succeeded:
return 0;

TypeCode Member Functions

A TypeCode represents OMG IDL type information.

No constructors for TypeCodes are defined. However, in addition to the mapped interface, for each basic and defined OMG IDL type, an implementation provides access to a TypeCode pseudo-object reference (TypeCode_ptr) of the form _tc_<type> that may be used to set types in Any, as arguments for equal, and so on. In the names of these TypeCode reference constants, <type> refers to the local name of the type within its defining scope. Each C++ _tc_<type> constant is defined at the same scoping level as its matching type.

Like all other serverless objects, the C++ mapping for TypeCode provides a _nil() operation that returns a nil object reference for a TypeCode. This operation can be used to initialize TypeCode references embedded within constructed types. However, a nil TypeCode reference may never be passed as an argument to an operation, since TypeCodes are effectively passed as values, not as object references.

The mapping of these member functions to C++ is as follows:

class CORBA
{
    class TypeCode
    {
        public:
class Bounds { ... };
class BadKind { ... };

Boolean equal(TypeCode_ptr) const;
TCKind kind() const;
Long param_count() const;
Any *parameter(Long) const;
RepositoryId id () const;
}; // TypeCode
}; // CORBA

Memory Management

TypeCode has the following special memory management rule:

- Ownership of the return values of the id function is maintained by the TypeCode; these return values must not be freed by the caller.

The following sections describe these member functions.

CORBA::TypeCode::equal

Synopsis
Determines whether two TypeCode objects are equal.

C++ Binding

CORBA::Boolean CORBA::TypeCode::equal (  
    CORBA::TypeCode_ptr     TypeCodeObj) const;

Argument

TypeCodeObj
A pointer to a TypeCode object with which to make the comparison.

Description
This member function determines whether a TypeCode object is equal to the input parameter, TypeCodeObj.
Return Values
If the TypeCode object is equal to the TypeCodeObj parameter, CORBA_TRUE is returned.
If the TypeCode object is not equal to the TypeCodeObj parameter, CORBA_FALSE is returned.
If the function does not succeed, an exception is thrown.

**CORBA::TypeCode::id**

**Synopsis**
Returns the ID for the TypeCode.

**C++ Binding**
```cpp
CORBA::RepositoryId CORBA::TypeCode::id () const;
```

**Arguments**
None.

**Description**
This member function returns the ID for the TypeCode.

**Return Values**
Repository ID for the TypeCode.

**CORBA::TypeCode::kind**

**Synopsis**
Retrieves the kind of data contained in the TypeCode object reference.

**C++ Binding**
```cpp
CORBA::TCKind CORBA::TypeCode::kind () const;
```

**Arguments**
None.

**Description**
This member function retrieves the kind attribute of the CORBA::TypeCode class, which specifies the kind of data contained in the TypeCode object reference.
Return Values

If the member function succeeds, it returns the kind of data contained in the TypeCode object reference. For a list of the TypeCode kinds and their parameters, see Table 14-2.

If the member function does not succeed, an exception is thrown.

Table 14-2  Legal Typecode Kinds and Parameters

<table>
<thead>
<tr>
<th>TypeCode Kind</th>
<th>Parameters List</th>
</tr>
</thead>
<tbody>
<tr>
<td>CORBA::tk_null</td>
<td><em>NONE</em></td>
</tr>
<tr>
<td>CORBA::tk_void</td>
<td><em>NONE</em></td>
</tr>
<tr>
<td>CORBA::tk_short</td>
<td><em>NONE</em></td>
</tr>
<tr>
<td>CORBA::tk_long</td>
<td><em>NONE</em></td>
</tr>
<tr>
<td>CORBA::tk_long</td>
<td><em>NONE</em></td>
</tr>
<tr>
<td>CORBA::tk_ushort</td>
<td><em>NONE</em></td>
</tr>
<tr>
<td>CORBA::tk_ulong</td>
<td><em>NONE</em></td>
</tr>
<tr>
<td>CORBA::tk_float</td>
<td><em>NONE</em></td>
</tr>
<tr>
<td>CORBA::tk_double</td>
<td><em>NONE</em></td>
</tr>
<tr>
<td>CORBA::tk_boolean</td>
<td><em>NONE</em></td>
</tr>
<tr>
<td>CORBA::tk_char</td>
<td><em>NONE</em></td>
</tr>
<tr>
<td>CORBA::tk_wchar</td>
<td><em>NONE</em></td>
</tr>
<tr>
<td>CORBA::tk_octet</td>
<td><em>NONE</em></td>
</tr>
<tr>
<td>CORBA::tk_Typecode</td>
<td><em>NONE</em></td>
</tr>
<tr>
<td>CORBA::tk_Principal</td>
<td><em>NONE</em></td>
</tr>
<tr>
<td>CORBA::tk_objref</td>
<td>{interface_id}</td>
</tr>
<tr>
<td>CORBA::tk_struct</td>
<td>{struct-name, member-name, TypeCode, ... (repeat pairs)}</td>
</tr>
<tr>
<td>CORBA::tk_union</td>
<td>{union-name, switch-TypeCode, label-value, member-name, enum-id, ...}</td>
</tr>
</tbody>
</table>
**Synopsis**
Retrieves the number of parameters for the TypeCode object reference.

**C++ Binding**
```cpp
CORBA::Long CORBA::TypeCode::param_count () const;
```

**Arguments**
None.

**Description**
This member function retrieves the parameter attribute of the `CORBA::TypeCode` class, which specifies the number of parameters for the TypeCode object reference. For a list of parameters of each kind, see Table 14-2.

**Return Values**
If the function succeeds, it returns the number of parameters contained in the TypeCode object reference.
If the function does not succeed, an exception is thrown.

### Table 14-2  Legal Typecode Kinds and Parameters (Continued)

<table>
<thead>
<tr>
<th>TypeCode Kind</th>
<th>Parameters List</th>
</tr>
</thead>
<tbody>
<tr>
<td>CORBA::tk_enum</td>
<td>{enum-name, enum-id, ...}</td>
</tr>
<tr>
<td>CORBA::tk_string</td>
<td>{maxlen-integer}</td>
</tr>
<tr>
<td>CORBA::tk_wstring</td>
<td>{maxlen-integer}</td>
</tr>
<tr>
<td>CORBA::tk_sequence</td>
<td>{TypeCode, maxlen-integer}</td>
</tr>
<tr>
<td>CORBA::tk_array</td>
<td>{TypeCode, length-integer}</td>
</tr>
</tbody>
</table>
**CORBA::TypeCode::parameter**

**Synopsis**
Retrieves a parameter specified by the index input argument.

**C++ Binding**

```c++
CORBA::Any * CORBA::TypeCode::parameter (CORBA::Long Index) const;
```

**Argument**

**Index**
An index to the parameter list, used to determine which parameter to retrieve.

**Description**
This member function retrieves a parameter specified by the index input argument. For a list of parameters of each kind, see Table 14-2.

**Return Values**
If the member function succeeds, the return value is a pointer to the parameter specified by the index input argument.

If the member function does not succeed, an exception is thrown.

**Exception Member Functions**
The BEA Tuxedo software supports the throwing and catching of exceptions.

**Caution:** Use of the wrong exception constructor causes noninitialization of a data member. Exceptions that are defined to have a `reason` field need to be created using the constructor that initializes that data member. If the default constructor is used instead, that data member is not initialized and, during destruction of the exception, the system may attempt to destroy nonexistent data.

When creating exceptions, be sure to use the constructor function that most fully initializes the data fields. These exceptions can be most easily identified by looking at the OMG IDL definition; they have additional data member definitions.

Descriptions of exception member functions follow:
Exception Member Functions

CORBA::SystemException::SystemException ()
This is the default constructor for the CORBA::SystemException class. Minor code is initialized to 0 (zero) and the completion status is set to COMPLETED_NO.

CORBA::SystemException::SystemException (const CORBA::SystemException & Se)
This is the copy constructor for the CORBA::SystemException class.

CORBA::SystemException::SystemException(CORBA::ULong Minor, CORBA::CompletionStatus Status)
This constructor for the CORBA::SystemException class sets the minor code and completion status.

Explanations of the arguments are as follows:

Minor
The minor code for the Exception object. The minor field is an implementation-specific value used by the ORB to identify the exception. The BEA Tuxedo minor field definitions can be found in the file orbminor.h.

Status
The completion status for the Exception object. The values are as follows:
CORBA::COMPLETED_YES
CORBA::COMPLETED_NO
CORBA::COMPLETED_MAYBE

CORBA::SystemException::~SystemException ()
This is the destructor for the CORBA::SystemException class. It frees any memory used for the Exception object.

CORBA::SystemException CORBA::SystemException::operator = (const CORBA::SystemException Se)
This assignment operator copies exception information from the source exception. The Se argument specifies the SystemException object that is to be copied by this operator.

CORBA::CompletionStatus CORBA::SystemException::completed()
This member function returns the completion status for this exception.

CORBA::SystemException::completed(
CORBA::CompletionStatus Completed)
This member function sets the completion status for this exception. The Completed argument specifies the completion status for this exception.

CORBA::ULong CORBA::SystemException::minor()
This member function returns the minor code for this exception.
CORBA::SystemException::minor (CORBA::ULong Minor)

This member function sets the minor code for this exception. The minor argument specifies the new minor code for this exception. The minor field is an implementation-specific value used by the application to identify the exception.

CORBA::SystemException * CORBA::SystemException::_narrow (CORBA::Exception_ptr Exc)

This member function determines whether a specified exception can be narrowed to a system exception. The Exc argument specifies the exception to be narrowed.

If the specified exception is a system exception, this member function returns a pointer to the system exception. If the specified exception is not a system exception, the function returns 0 (zero).

CORBA::UserException * CORBA::UserException::_narrow (CORBA::Exception_ptr Exc)

This member function determines whether a specified exception can be narrowed to a user exception. The Exc argument specifies the exception to be narrowed.

If the specified exception is a user exception, this member function returns a pointer to the user exception. If the specified exception is not a user exception, the function returns 0 (zero).

**Standard Exceptions**

This section presents the standard exceptions defined for the ORB. These exception identifiers may be returned as a result of any operation invocation, regardless of the interface specification. Standard exceptions are not listed in raises expressions.

To bound the complexity in handling the standard exceptions, the set of standard exceptions is kept to a tractable size. This constraint forces the definition of equivalence classes of exceptions, rather than enumerating many similar exceptions.

For example, an operation invocation can fail at many different points due to the inability to allocate dynamic memory. Rather than enumerate several different exceptions that correspond to the different ways that memory allocation failure causes the exception (during marshaling, unmarshaling, in the client, in the object implementation, allocating network packets, and so forth), a single exception corresponding to dynamic memory allocation failure is defined. Each standard exception includes a minor code to designate the subcategory of the exception; the assignment of values to the minor codes is left to each ORB implementation.

Each standard exception also includes a completion_status code, which takes one of the following values:
CORBA::COMPLETED_YES
The object implementation completed processing prior to the exception being raised.

CORBA::COMPLETED_NO
The object implementation was not initiated prior to the exception being raised.

CORBA::COMPLETED_MAYBE
The status of implementation completion is unknown.

Exception Definitions
The standard exceptions are defined below. Clients must be prepared to handle system exceptions that are not on this list, both because future versions of this specification may define additional standard exceptions, and because ORB implementations may raise nonstandard system exceptions. For more information about exceptions, see System Messages.

Table 14-3 defines the exceptions.

<table>
<thead>
<tr>
<th>Exception</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CORBA::UNKNOWN</td>
<td>The unknown exception.</td>
</tr>
<tr>
<td>CORBA::BAD_PARAM</td>
<td>An invalid parameter was passed.</td>
</tr>
<tr>
<td>CORBA::NO_MEMORY</td>
<td>Dynamic memory allocation failure.</td>
</tr>
<tr>
<td>CORBA::IMP_LIMIT</td>
<td>Violated implementation limit.</td>
</tr>
<tr>
<td>CORBA::COMM_FAILURE</td>
<td>Communication failure.</td>
</tr>
<tr>
<td>CORBA::INV_OBJREF</td>
<td>Invalid object reference.</td>
</tr>
<tr>
<td>CORBA::NO_PERMISSION</td>
<td>No permission for attempted operation.</td>
</tr>
<tr>
<td>CORBA::INTERNAL</td>
<td>ORB internal error.</td>
</tr>
<tr>
<td>CORBA::MARSHAL</td>
<td>Error marshalling parameter/result.</td>
</tr>
<tr>
<td>CORBA::INITIALIZE</td>
<td>ORB initialization failure.</td>
</tr>
<tr>
<td>CORBA::NO_IMPLEMENT</td>
<td>Operation implementation unavailable.</td>
</tr>
<tr>
<td>CORBA::BAD_TYPECODE</td>
<td>Bad typecode.</td>
</tr>
</tbody>
</table>
Object Nonexistence

The `CORBA::OBJECT_NOT_EXIST` exception is raised whenever an invocation on a deleted object is performed. It is an authoritative “hard” fault report. Anyone receiving it is allowed (even expected) to delete all copies of this object reference and to perform other appropriate “final recovery” style procedures.

### Table 14-3 Exception Definitions (Continued)

<table>
<thead>
<tr>
<th>Exception</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CORBA::BAD_OPERATION</td>
<td>Invalid operation.</td>
</tr>
<tr>
<td>CORBA::NO_RESOURCES</td>
<td>Insufficient resources for request.</td>
</tr>
<tr>
<td>CORBA::NO_RESPONSE</td>
<td>Response to request not yet available.</td>
</tr>
<tr>
<td>CORBA::PERSIST_STORE</td>
<td>Persistent storage failure.</td>
</tr>
<tr>
<td>CORBA::BAD_INV_ORDER</td>
<td>Routine invocations out of order.</td>
</tr>
<tr>
<td>CORBA::TRANSIENT</td>
<td>Transient failure; reissue request.</td>
</tr>
<tr>
<td>CORBA::FREE_MEM</td>
<td>Cannot free memory.</td>
</tr>
<tr>
<td>CORBA::INV_IDENT</td>
<td>Invalid identifier syntax.</td>
</tr>
<tr>
<td>CORBA::INV_FLAG</td>
<td>Invalid flag was specified.</td>
</tr>
<tr>
<td>CORBA::INTF_REPOS</td>
<td>Error accessing interface repository.</td>
</tr>
<tr>
<td>CORBA::BAD_CONTEXT</td>
<td>Error processing context object.</td>
</tr>
<tr>
<td>CORBA::OBJ_ADAPTER</td>
<td>Failure detected by object adapter.</td>
</tr>
<tr>
<td>CORBA::DATA_CONVERSION</td>
<td>Data conversion error.</td>
</tr>
<tr>
<td>CORBA::OBJECT_NOT_EXIST</td>
<td>Nonexistent object; delete reference.</td>
</tr>
<tr>
<td>CORBA::TRANSACTION_REQUIRED</td>
<td>Transaction required.</td>
</tr>
<tr>
<td>CORBA::TRANSACTION_ROLLEDBACK</td>
<td>Transaction rolled back.</td>
</tr>
<tr>
<td>CORBA::INVALID_TRANSACTION</td>
<td>Invalid transaction.</td>
</tr>
</tbody>
</table>
Transaction Exceptions

The CORBA::TRANSACTION_REQUIRED exception indicates that the request carried a NULL transaction context, but an active transaction is required.

The CORBA::TRANSACTION_ROLLEDBACK exception indicates that the transaction associated with the request has already been rolled back or marked to roll back. Thus, the requested operation either could not be performed or was not performed because further computation on behalf of the transaction would be fruitless.

The CORBA::INVALID_TRANSACTION indicates that the request carried an invalid transaction context. For example, this exception could be raised if an error occurred when trying to register a resource.

ExceptionList Member Functions

The ExceptionList member functions allow a client or server application to provide a list of TypeCodes for all user-defined exceptions that may result when the Request is invoked. For a description of the Request member functions, see the section Request Member Functions.

The mapping of these member functions to C++ is as follows:

class CORBA
{
    class ExceptionList
    {
        public:
            Ulong count () const;
            void add(TypeCode_ptr tc);
            void add_consume(TypeCode_ptr tc);
            TypeCode_ptr item(Ulong index);
            Status remove(Ulong index);
    }; // ExceptionList
}; // CORBA
CORBA::ExceptionList::count

Synopsis
Retrieves the current number of items in the list.

C++ Binding
Ulong count ();

Arguments
None.

Exception
If the function does not succeed, an exception is thrown.

Description
This member function retrieves the current number of items in the list.

Return Values
If the function succeeds, the returned value is the number of items in the list. If the list has just been created, and no ExceptionList objects have been added, this function returns 0 (zero).

CORBA::ExceptionList::add

Synopsis
Constructs a ExceptionList object with an unnamed item, setting only the flags attribute.

C++ Binding
void add(TypeCode_ptr tc);

Arguments
tc
Defines the memory location referred to by TypeCode_ptr.

Exception
If the member function does not succeed, a CORBA::NO_MEMORY exception is thrown.
Description
This member function constructs an ExceptionList object with an unnamed item, setting only the flags attribute.
The ExceptionList object grows dynamically; your application does not need to track its size.

Return Values
If the function succeeds, the return value is a pointer to the newly created ExceptionList object.

See Also
CORBA::ExceptionList::add_consume
CORBA::ExceptionList::count
CORBA::ExceptionList::item
CORBA::ExceptionList::remove

**CORBA::ExceptionList::add_consume**

Synopsis
Constructs an ExceptionList object.

C++ Binding

```cpp
void add_consume(TypeCode_ptr tc);
```

Arguments

- `tc`
  The memory location to be assumed.

Exceptions
If the member function does not succeed, an exception is raised.

Description
This member function constructs an ExceptionList object.
The ExceptionList object grows dynamically; your application does not need to track its size.

Return Values
If the function succeeds, the return value is a pointer to the newly created ExceptionList object.
See Also

CORBA::ExceptionList::add
CORBA::ExceptionList::count
CORBA::ExceptionList::item
CORBA::ExceptionList::remove

**CORBA::ExceptionList::item**

**Synopsis**

Retrieves a pointer to the ExceptionList object, based on the index passed in.

**C++ Binding**

```
TypeCode_ptr item(ULong index);
```

**Argument**

`index`

The index into the ExceptionList object. The indexing is zero-based.

**Exceptions**

If the function does not succeed, the **BAD_PARAM** exception is thrown.

**Description**

This member function retrieves a pointer to an ExceptionList object, based on the index passed in. The function uses zero-based indexing.

**Return Values**

If the function succeeds, the return value is a pointer to the ExceptionList object.

**See Also**

CORBA::ExceptionList::add
CORBA::ExceptionList::add_consume
CORBA::ExceptionList::count
CORBA::ExceptionList::remove
CORBA::ExceptionList::remove

**Synopsis**

Removes the item at the specified index, frees any associated memory, and reorders the remaining items on the list.
C++ Binding

    Status remove(ULong index);

Argument

Index
    The index into the ContextList object. The indexing is zero-based.

Exceptions

If the function does not succeed, the BAD_PARAM exception is thrown.

Description

This member function removes the item at the specified index, frees any associated memory, and reorders the remaining items on the list.

Return Values

None.

See Also

    CORBA::ExceptionList::add
    CORBA::ExceptionList::add_consume
    CORBA::ExceptionList::count
    CORBA::ExceptionList::item
Server-side mapping refers to the portability constraints for an object implementation written in C++. The term *server* is not meant to restrict implementations to situations in which method invocations cross-address space or machine boundaries. This mapping addresses any implementation of an Object Management Group (OMG) Interface Definition Language (IDL) interface.

**Note:** The information in this chapter is based on the *Common Object Request Broker: Architecture and Specification*, Revision 2.4.2, February 2001, published by the Object Management Group (OMG). Used with permission of the OMG.

## Implementing Interfaces

To define an implementation in C++, you define a C++ class with any valid C++ name. For each operation in the interface, the class defines a nonstatic member function with the mapped name of the operation (the mapped name is the same as the OMG IDL identifier).

The server application mapping specifies two alternative relationships between the implementation class supplied by the application and the generated class or classes for the interface. Specifically, the mapping requires support for both inheritance-based relationships and delegation-based relationships. Conforming applications may use either or both of these alternatives. BEA Tuxedo CORBA supports both inheritance-based and delegation-based relationships.
Inheritance-based Interface Implementation

In the inheritance-based interface implementation approach, the implementation classes are derived from a generated base class based on the OMG IDL interface definition. The generated base classes are known as skeleton classes, and the derived classes are known as implementation classes. Each operation of the interface has a corresponding virtual member function declared in the skeleton class. The generated skeleton class is partially opaque to the programmer, though it will contain a member function corresponding to each operation in the interface. The signature of the member function is identical to that of the generated client stub class.

To implement this interface using inheritance, a programmer must derive from this skeleton class and implement each of the operations in the OMG IDL interface. To allow portable implementations to multiple inheritances from both skeleton classes and implementation classes for other base interfaces without error or ambiguity, the Tobj_ServantBase class must be a virtual base class of the skeleton, and the PortableServer::ServantBase class must be a virtual base class of the Tobj_ServantBase class. The inheritance among the implementation class, the skeleton class, the Tobj_ServantBase class, and the PortableServer::ServantBase class must all be public virtual.

The implementation class or servant must only derive directly from a single generated skeleton class. Direct derivation from multiple skeleton classes could result in ambiguous errors due to multiple definitions of the _this() operation. This should not be a limitation, however, since CORBA objects have only a single most-derived interface. C++ servants that are intended to support multiple interface types can utilize the delegation-based interface implementation approach. See Listing 15-1 for an example of OMG IDL that uses interface inheritance.

Listing 15-1  OMG IDL That Uses Interface Inheritance

```idl
// IDL
interface A
{
    short op1();
    void op2(in long val);
};
```
Inheritance-based Interface Implementation

Listing 15-2  Interface Class A

```cpp
// C++
class A : public virtual CORBA::Object
{
    public:
        virtual CORBA::Short op1();
        virtual void op2(CORBA::Long val);
};
```

On the server side, a skeleton class is generated. This class is partially opaque to the programmer, though it does contain a member function corresponding to each operation in the interface.

For the Portable Object Adapter (POA), the name of the skeleton class is formed by prepending the string “POA_” to the fully scoped name of the corresponding interface, and the class is directly derived from the servant base class `Tobj_ServantBase`. The C++ mapping for `Tobj_ServantBase` is as follows:

```cpp
// C++
class Tobj_ServantBase
{
    public:
        virtual void activate_object(const char* stroid);
        virtual void deactivate_object(const char* stroid,  
                                        TobjS::DeactivateReasonValue reason);
};
```

The `activate_object()` and `deactivate_object()` member functions are described in detail in the sections `Tobj_ServantBase::activate_object()` and `Tobj_ServantBase::_add_ref()`.

The skeleton class for interface A shown above would appear as shown in Listing 15-3.

Listing 15-3  Skeleton Class for Interface A

```cpp
// C++
class POA_A : public Tobj_ServantBase
```
If interface A were defined within a module rather than at global scope (for example, \texttt{Mod::A}), the name of its skeleton class would be \texttt{POA\_Mod::A}. This helps to separate server application skeleton declarations and definitions from C++ code generated for the client.

To implement this interface using inheritance, you must derive from this skeleton class and implement each of the operations in the corresponding OMG IDL interface. An implementation class declaration for interface A would take the form shown in Listing 15-4.

### Listing 15-4  Interface A Implementation Class Declaration

```cpp
// C++
class A_impl : public POA_A
{
  public:
    CORBA::Short op1();
    void op2(CORBA::Long val);
    ...
};
```

### Delegation-based Interface Implementation

The delegation-based interface implementation approach is an alternative to using inheritance when implementing CORBA objects. This approach is used when the overhead of inheritance is too high or cannot be used. For example, due to the invasive nature of inheritance, implementing objects using existing legacy code might be impossible if inheritance for some global class were...
required. Instead, delegation can be used to solve these types of problems. Delegation is a more natural fit doing object implementations when the Process-Entity design pattern is used. In this pattern, the Process object would delegate operations onto one or more entity objects.

In the delegation-based approach, the implementation does not inherit from a skeleton class. Instead, the implementation can be coded as required for the application, and a wrapper object will delegate upcalls to that implementation. This “wrapper object,” called a tie, is generated by the IDL compiler, along with the same skeleton class used for the inheritance approach. The generated tie class is partially opaque to the programmer, though, like the skeleton, it provides a method corresponding to each OMG IDL operation for the associated interface. The name of the generated tie class is the same as the generated skeleton class with the addition that the string _tie is appended to the end of the class name.

An instance of the tie class is the servant, not the C++ object being delegated to by the tie object, that is passed as the argument to the operations that require a Servant argument. It should also be noted that the tied object has no access to the _this operation, nor should it access data members directly.

A type-safe tie class is implemented using C++ templates. The code shown in Listing 15-5 illustrates a tie class generated from the Derived interface in the previous OMG IDL example.

**Listing 15-5  Tie Class Generated from the Derived Interface**

```cpp
// C++
template <class T>
class POA_A_tie : public POA_A {
public:
  POA_A_tie(T& t)
    : _ptr(&t), _poa(PortableServer::POA::nil()), _rel(0) {}
  POA_A_tie(T& t, PortableServer::POA_ptr poa)
    : _ptr(&t), _poa(PortableServer::POA::duplicate(poa)), _rel(0) {}
  POA_A_tie(T* tp, CORBA::Boolean release = 1)
    : _ptr(tp), _poa(PortableServer::POA::nil()), _rel(release) {}
  POA_A_tie(T* tp, PortableServer::POA_ptr poa, CORBA::Boolean release = 1)
    : _ptr(tp), _poa(PortableServer::POA::duplicate(poa)), _rel(release) {}
  ~POA_A_tie() { CORBA::release(_poa);
    if (_rel) delete _ptr;
  }

  // tie-specific functions
  T* _tied_object() { return _ptr; }
  void _tied_object(T& obj)
```

Delegation-based Interface Implementation
{ if (_rel) delete _ptr;
  _ptr = &obj;
  _rel = 0;
}

void _tied_object(T* obj, CORBA::Boolean release = 1)
{ if (_rel) delete _ptr;
  _ptr = obj;
  _rel = release;
}

CORBA::Boolean _is_owner() { return _rel; }
void _is_owner(CORBA::Boolean b) { _rel = b; }

// IDL operations********************************************************************************
CORBA::Short op1 ()
{    return _ptr->op1 ();
}

void op2 (CORBA::Long val)
{    _ptr->op2 (val);
}  // ***************************************************************

// override ServantBase operations
PortableServer::POA_ptr _default_POA()
{
  if (!CORBA::is_nil(_poa))
  {
    return _poa;
  }
  else {
    #ifdef WIN32
      return ServantBase::default_POA();
    #else
      return PortableServer::ServantBase::default_POA();
    #endif
  }
}

private:
  T* _ptr;
  PortableServer::POA_ptr _poa;
  CORBA::Boolean _rel;

  // copy and assignment not allowed
POA_A_tie (const POA_A_tie<T> &);
void operator=(const POA_A_tie<T> &);

This class definition is a template generated by the IDL compiler. You typically use it by first getting a pointer to the legacy class and then instantiating the tie class with that pointer. For example:

```cpp
Old::Legacy * legacy = new Old::Legacy( oid);
POA_A_tie<Old::Legacy> * A_servant_ptr =
    new POA_A_tie<Old::Legacy>( legacy);
```

As you can see, the tie class contains definitions for the op1 and op2 operations of the interface that assume that the legacy class has operations with the same signatures as those given in the IDL. If this is the case, you can use the tie class file as is, letting it delegate exactly. It is more likely, however, that the legacy class will not have identical signatures or you may have to do more than a single function call. In that case, it is your job to replace the code for op1 and op2 in this generated code. The code for each operation typically makes invocations on the legacy class using the tie class variable _ptr, which contains the pointer to the legacy class. For example, you might change the following lines:

```cpp
CORBA::Short op1 () {return _ptr->op1 (); }
void op2 (CORBA::Long val) {_ptr->op2 (val); }
```

to the following:

```cpp
CORBA::Short op1 ()
{
    return _ptr->op37 ();
}

void op2 (CORBA::Long val)
{
    CORBA::Long temp;
    temp = val + 15;
    _ptr->lookup(val, temp, 43);
}
```

An instance of this template class performs the task of delegation. When the template is instantiated with a class type that provides the operation of the Derived interface, then the
POA_Derived_tie class will delegate all operations to an instance of that implementation class. A reference or pointer to the actual implementation object is passed to the appropriate tie constructor when an instance of the POA_Derived_tie class is created. When a request is invoked on it, the tie servant will just delegate the request by calling the corresponding method on the implementation class.

The use of templates for tie classes allows the application developer to provide specializations for some or all of the template’s operations for a given instantiation of the template. This allows the application to use legacy classes for tied object types, where the operation signatures of the tied object will differ from that of the tie class.

**Implementing Operations**

The signature of an implementation member function is the mapped signature of the OMG IDL operation. Unlike the client-side mapping, the OMG specifies that the function header for the server-side mapping include the appropriate exception specification. An example of this is shown in Listing 15-6.

**Listing 15-6 Exception Specification**

```idl
// IDL
interface A
{
  exception B {};
  void f() raises(B);
};
```

```cpp
// C++
class MyA : public virtual POA_A
{
  public:
    void f();
    ...
};
```
Since all operations and attributes may raise CORBA system exceptions, CORBA::SystemException must appear in all exception specifications, even when an operation has no raises clause.

**Note:** Because of the differences in C++ compilers, it is best to leave out the "throw declaration" in the method signature. Some systems cause the application server to crash if an undeclared exception is thrown in a method that has declared the exceptions it will throw.

Within a member function, the “this” pointer refers to the implementation object’s data as defined by the class. In addition to accessing the data, a member function may implicitly call another member function defined by the same class. An example of this is shown in Listing 15-7.

### Listing 15-7  Calling Another Member Function

```cpp
// IDL
interface A
{
    void f();
    void g();
};

// C++
class MyA : public virtual POA_A
{
    public:
        void f();
        void g();
    private:
        long x_;  
};

void MyA::f();
{
    x_ = 3;
    g();
}
```
When a servant member function is invoked in this manner, it is being called simply as a C++
member function, not as the implementation of an operation on a CORBA object.