

WebSphere Application Server CORBA support

CORBA support (Reference articles)

Note!

Before using this information and the product it supports, be sure to read the general information under "Notices" on page 308 .

Contents

| | |
|---|----|
| CORBA support reference articles | 1 |
| Interface Definition Language (IDL) | 2 |
| IDL name scoping | 2 |
| IDL interface declarations | 3 |
| IDL constant declarations | 4 |
| IDL type declarations | 4 |
| IDL exception declarations | 7 |
| IDL attribute declarations | 8 |
| IDL operation declarations | 8 |
| Multiple IDL interfaces | 10 |
| IDL include directives | 11 |
| IDL pragma directives | 11 |
| The idlc command (IDL compiler) | 13 |
| Options for the idlc command | 14 |
| The IDL-to-Java compiler | 17 |
| Options for the IDL-to-Java compiler | 17 |
| | |
| Conventions used in documenting IDL syntax | 21 |
| IDL lexical rules | 21 |
| IDL reserved words | 22 |
| Syntax for comments in IDL code | 22 |
| The implementation registration utility (regimpl) | 23 |
| CORBA programming | 24 |
| CORBA C++ bindings | 24 |
| CORBA C++ bindings for constants | 24 |
| | |
| CORBA types and business objects | 24 |
| | |
| CORBA C++ bindings for data types | 26 |
| | |
| CORBA C++ binding restrictions | 39 |
| CORBA programming: Name scoping and modules in the C++ bindings | 39 |
| Commonly used CORBA interfaces | 40 |
| CORBA class interfaces | 40 |
| CORBA::object interfaces | 40 |
| CORBA::ORB interfaces | 41 |
| C++ bindings for CORBA interfaces | 41 |
| Managing CORBA object references | 43 |
| | |
| Widening CORBA object references | 43 |
| | |
| Narrowing CORBA object references | 43 |
| | |
| CORBA programming: narrowing to a C++ implementation | 44 |
| CORBA programming: Storage management and _var types | 44 |
| CORBA programming: Argument passing considerations for C++ bindings | 45 |
| | |
| CORBA internationalization considerations | 51 |
| | |
| CORBA internationalization: Initializing client programs | 51 |

| | |
|---|----|
| CORBA internationalization: Character set restriction | 51 |
| CORBA internationalization: Passing object references over multiple platforms | 51 |
| | |
| CORBA internationalization: Using the OMG char data type in IDL files | 51 |
| The CORBA module | 52 |
| CORBA module: Types | 52 |
| CORBA module: AliasDef Interface | 54 |
| AliasDef::original_type_def | 54 |
| CORBA module: Any Class | 55 |
| Any::_nil | 57 |
| Any::operator<< | 57 |
| Any::operator>> | 58 |
| Any::replace | 60 |
| Any::type | 60 |
| CORBA module: ArrayDef Interface | 61 |
| ArrayDef::element_type | 62 |
| ArrayDef::element_type_def | 62 |
| ArrayDef::length | 63 |
| CORBA module: AttributeDef Interface | 64 |
| | |
| AttributeDef::describe | 64 |
| AttributeDef::mode | 65 |
| AttributeDef::type_def | 66 |
| CORBA module: BOA Class | 67 |
| BOA::_duplicate | 67 |
| BOA::_nil | 68 |
| BOA::create | 68 |
| BOA::deactivate_impl | 69 |
| BOA::dispose | 70 |
| BOA::execute_next_request | 71 |
| BOA::execute_request_loop | 71 |
| BOA::get_id | 72 |
| BOA::get_principal | 73 |
| BOA::impl_is_ready | 73 |
| BOA::request_pending | 74 |
| CORBA module: | 75 |
| BOA::DynamicImplementation Class | 75 |
| BOA::DynamicImplementation::invoke | 75 |
| CORBA module: ConstantDef Interface | 76 |
| | |
| ConstantDef::describe | 77 |
| ConstantDef::type_def | 78 |
| ConstantDef::value | 78 |
| CORBA module: Contained Interface | 79 |
| Contained::absolute_name | 80 |
| Contained::containing_repository | 81 |
| Contained::defined_in | 81 |
| Contained::describe | 82 |
| Contained::id | 83 |
| Contained::name | 84 |
| Contained::version | 84 |
| CORBA module: Container Interface | 85 |
| Container::contents | 86 |
| Container::create_alias | 87 |
| Container::create_constant | 88 |

Contents

| | | | |
|---|-----|--|-----|
| Container::create_enum | 89 | ExceptionList::add | 125 |
| Container::create_exception | 90 | ExceptionList::add_consume | 125 |
| Container::create_interface | 91 | ExceptionList::count | 125 |
| Container::create_module | 92 | ExceptionList::item | 126 |
| Container::create_struct | 93 | ExceptionList::remove | 126 |
| Container::create_union | 94 | CORBA module: IDLType Interface | 127 |
| Container::describe_contents | 95 | IDLType::type | 127 |
| Container::lookup | 96 | CORBA module: ImplementationDef | 128 |
| Container::lookup_name | 97 | Interface | |
| CORBA module: Context Class | 98 | ImplementationDef::get_alias | 129 |
| Context::_duplicate | 99 | ImplementationDef::get_id | 129 |
| Context::_nil | 99 | CORBA module: ImplRepository Class | 130 |
| Context::context_name | 99 | | |
| Context::create_child | 100 | ImplRepository::find_impldef | 130 |
| Context::delete_values | 100 | ImplRepository::find_impldef_by_alias | 131 |
| Context::get_values | 101 | | |
| Context::parent | 101 | CORBA module: InterfaceDef Interface | 132 |
| Context::set_one_value | 102 | | |
| Context::set_values | 102 | InterfaceDef::base_interfaces | 133 |
| CORBA module: ContextList Class | 103 | InterfaceDef::create_attribute | 134 |
| ContextList::_duplicate | 103 | InterfaceDef::create_operation | 135 |
| ContextList::_nil | 104 | InterfaceDef::describe | 136 |
| ContextList::add | 104 | InterfaceDef::describe_interface | 137 |
| ContextList::add_consume | 104 | InterfaceDef::is_a | 138 |
| ContextList::count | 105 | CORBA module: IRObjct Interface | 138 |
| ContextList::item | 105 | IRObjct::def_kind | 139 |
| ContextList::remove | 106 | IRObjct::destroy | 139 |
| CORBA module: CORBA Class | 106 | CORBA module: ModuleDef Interface ... | 140 |
| CORBA::_boa | 109 | ModuleDef::describe | 140 |
| CORBA::is_nil | 109 | CORBA module: NamedValue Class | 141 |
| CORBA::ORB_init | 110 | NamedValue::_duplicate | 142 |
| CORBA::release | 111 | NamedValue::_nil | 142 |
| CORBA::string_alloc | 112 | NamedValue::flags | 142 |
| CORBA::string_dup | 113 | NamedValue::name | 143 |
| CORBA::string_free | 113 | NamedValue::value | 143 |
| CORBA::wstring_alloc | 114 | CORBA module: NVList Class | 144 |
| CORBA::wstring_dup | 114 | NVList::_duplicate | 144 |
| CORBA::wstring_free | 115 | NVList::_nil | 145 |
| CORBA module: Current Class | 115 | NVList::add | 145 |
| Current::_duplicate | 116 | NVList::add_item | 146 |
| Current::_nil | 116 | NVList::add_item_consume | 146 |
| CORBA module: EnumDef Interface | 116 | NVList::add_value | 147 |
| EnumDef::members | 117 | NVList::add_value_consume | 148 |
| CORBA module: Environment Class | 118 | NVList::count | 148 |
| Environment::_duplicate | 118 | NVList::get_item_index | 149 |
| Environment::_nil | 118 | NVList::item | 149 |
| Environment::clear | 119 | NVList::remove | 150 |
| Environment::exception | 119 | CORBA module: Object Class | 150 |
| CORBA module: Exception Class | 120 | Object::_create_request | 151 |
| Exception::_duplicate | 120 | Object::_duplicate | 153 |
| Exception::_nil | 120 | Object::_get_implementation | 153 |
| Exception::id | 121 | Object::_get_interface | 154 |
| CORBA module: ExceptionDef Interface | 121 | Object::_hash | 155 |
| | | Object::_is_a | 155 |
| ExceptionDef::describe | 122 | Object::_is_equivalent | 156 |
| ExceptionDef::members | 122 | Object::_narrow | 157 |
| CORBA module: ExceptionList Class | 123 | Object::_nil | 157 |
| ExceptionList::_duplicate | 124 | Object::_non_existent | 158 |
| ExceptionList::_nil | 124 | Object::_request | 158 |

Contents

| | | | |
|--|-----|---|-----|
| Object::_this | 159 | Request::_nil | 197 |
| CORBA module: OperationDef Interface | 160 | Request::add_in_arg | 198 |
| | | Request::add_inout_arg | 198 |
| OperationDef::contexts | 161 | Request::add_out_arg | 199 |
| OperationDef::describe | 162 | Request::arguments | 199 |
| OperationDef::exceptions | 163 | Request::contexts | 200 |
| OperationDef::mode | 163 | Request::ctx | 200 |
| OperationDef::params | 164 | Request::env | 201 |
| OperationDef::result | 165 | Request::exceptions | 201 |
| OperationDef::result_def | 166 | Request::get_response | 202 |
| CORBA module: ORB Class | 167 | Request::invoke | 202 |
| ORB::_duplicate | 168 | Request::operation | 202 |
| ORB::_nil | 168 | Request::poll_response | 203 |
| ORB::BOA_init | 169 | Request::result | 203 |
| ORB::create_alias_tc | 170 | Request::return_value | 204 |
| ORB::create_array_tc | 171 | Request::send_deferred | 204 |
| ORB::create_context_list | 171 | Request::send_oneway | 204 |
| ORB::create_enum_tc | 172 | Request::set_return_type | 205 |
| ORB::create_environment | 173 | Request::target | 205 |
| ORB::create_exception_list | 173 | CORBA module: RequestSeq Class | 206 |
| ORB::create_exception_tc | 174 | RequestSeq::allocbuf | 206 |
| ORB::create_interface_tc | 175 | RequestSeq::freebuf | 206 |
| ORB::create_list | 175 | RequestSeq::length | 207 |
| ORB::create_named_value | 176 | RequestSeq::maximum | 207 |
| ORB::create_operation_list | 177 | RequestSeq::operator[] | 208 |
| ORB::create_recursive_sequence_tc | 177 | CORBA module: SequenceDef Interface | 208 |
| | | | |
| ORB::create_sequence_tc | 178 | SequenceDef::bound | 209 |
| ORB::create_string_tc | 179 | SequenceDef::element_type | 209 |
| ORB::create_struct_tc | 180 | SequenceDef::element_type_def | 210 |
| ORB::create_union_tc | 180 | CORBA module: ServerRequest Class | 211 |
| ORB::get_default_context | 181 | | |
| ORB::get_next_response | 182 | ServerRequest::_duplicate | 211 |
| ORB::get_service_information | 183 | ServerRequest::_nil | 212 |
| ORB::list_initial_services | 184 | ServerRequest::ctx | 212 |
| ORB::object_to_string | 184 | ServerRequest::exception | 213 |
| ORB::poll_next_response | 185 | ServerRequest::op_def | 213 |
| ORB::resolve_initial_references | 186 | ServerRequest::op_name | 213 |
| ORB::resolve_initial_references_remote | 186 | ServerRequest::params | 214 |
| | | ServerRequest::result | 215 |
| ORB::send_multiple_requests_deferred | 188 | CORBA module: StringDef Interface | 215 |
| | | StringDef::bound | 216 |
| ORB::send_multiple_requests_oneway | 188 | CORBA module: StructDef Interface | 216 |
| | | StructDef::members | 217 |
| ORB::string_to_object | 189 | CORBA module: SystemException Class | 218 |
| CORBA module: Policy Interface | 189 | | |
| CORBA module: PrimitiveDef Interface | 190 | SystemException::_duplicate | 219 |
| | | SystemException::_nil | 219 |
| PrimitiveDef::kind | 190 | SystemException::completed | 220 |
| CORBA module: Principal Interface | 191 | SystemException::minor | 220 |
| CORBA module: Repository Interface | 191 | CORBA module: TypeCode Class | 221 |
| Repository::create_array | 192 | TypeCode::_duplicate | 222 |
| Repository::create_sequence | 192 | TypeCode::_nil | 222 |
| Repository::create_string | 193 | TypeCode::content_type | 222 |
| Repository::create_wstring | 194 | TypeCode::default_index | 223 |
| Repository::get_primitive | 194 | TypeCode::discriminator_type | 223 |
| Repository::lookup_id | 195 | TypeCode::equal | 224 |
| CORBA module: Request Class | 195 | TypeCode::id | 224 |
| Request::_duplicate | 197 | TypeCode::kind | 225 |

Contents

| | | | |
|--|-----|--|-----|
| TypeCode::length | 225 | Coordinator::hash_top_level_transaction | 258 |
| TypeCode::member_count | 225 | Coordinator::hash_transaction | 258 |
| TypeCode::member_label | 226 | Coordinator::is_ancestor_transaction | 259 |
| TypeCode::member_name | 226 | Coordinator::is_descendant_transaction | 260 |
| TypeCode::member_type | 227 | Coordinator::is_related_transaction | 260 |
| TypeCode::name | 227 | Coordinator::is_same_transaction | 261 |
| CORBA module: TypedefDef Interface | 228 | Coordinator::is_top_level_transaction | 262 |
| TypedefDef::describe | 228 | Coordinator::register_resource | 263 |
| CORBA module: UnionDef Interface | 229 | Coordinator::register_subtran_aware | 263 |
| UnionDef::discriminator_type | 230 | Coordinator::register_synchronization | 263 |
| UnionDef::discriminator_type_def | 230 | Coordinator::rollback_only | 263 |
| UnionDef::members | 231 | Current::begin | 264 |
| CORBA module: UnknownUserException Class | 232 | Current::commit | 265 |
| UnknownUserException::_duplicate | 232 | Current::get_control | 267 |
| UnknownUserException::_nil | 233 | Current::get_status | 268 |
| UnknownUserException::exception | 233 | Current::get_transaction_name | 269 |
| CORBA module: UserException Class | 234 | Current::resume | 270 |
| UserException::_duplicate | 234 | Current::rollback | 271 |
| UserException::_nil | 234 | Current::rollback_only | 273 |
| CORBA module: WstringDef Interface | 235 | Current::set_timeout | 274 |
| WstringDef::bound | 235 | Current::suspend | 275 |
| CosNaming in the Naming Service | 237 | CosTransactions::RecoveryCoordinator Interface | 276 |
| CosNaming::BindingIterator Interface | 237 | CosTransactions::Resource Interface | 276 |
| BindingIterator::destroy | 238 | CosTransactions::Synchronization Interface | 276 |
| BindingIterator::next_n | 241 | Synchronization::after_completion | 276 |
| BindingIterator::next_one | 242 | Synchronization::before_completion | 277 |
| CosNaming::NamingContext Interface | 242 | CosTransactions::Terminator Interface | 277 |
| NamingContext::bind | 243 | Terminator::commit | 278 |
| NamingContext::bind_context | 244 | Terminator::rollback | 279 |
| NamingContext::bind_new_context | 245 | CosTransactions::TransactionalObject Interface | 280 |
| NamingContext::destroy | 245 | CosTransactions::TransactionFactory Interface | 280 |
| NamingContext::list | 246 | C++ value type library, methods implemented | 281 |
| NamingContext::new_context | 247 | Runtime properties for CORBA clients and servers | 289 |
| NamingContext::rebind | 247 | Reference information for problem determination | 298 |
| NamingContext::rebind_context | 248 | Description of a formatted activity log entry | 298 |
| NamingContext::resolve | 248 | CORBA system exception minor codes | 301 |
| NamingContext::unbind | 249 | | |
| CosTransactions in the Transaction Service | 251 | | |
| CosTransactions::Control Interface | 252 | | |
| Control::get_coordinator | 252 | | |
| Control::get_terminator | 253 | | |
| CosTransactions::Coordinator Interface | 254 | | |
| Coordinator::get_parent_status | 255 | | |
| Coordinator::get_status | 256 | | |
| Coordinator::get_top_level_status | 256 | | |
| Coordinator::get_transaction_name | 257 | | |
| Coordinator::get_txcontext | 258 | | |

CORBA support reference articles

This part contains reference topics about the CORBA support provided by WebSphere Application Server 4.0 enterprise services. These topics are intended to provide reference information that provides extra detailed information relevant to CORBA support that support the concept and task articles.

- [“CORBA support concept articles” on page](#)
- [“CORBA support example articles” on page](#)
- [“CORBA support task articles” on page](#)

Interface Definition Language (IDL)

The Object Management Group (OMG) Interface Definition Language (IDL) is the formal language used to define object interfaces independent of the programming language used to implement the those methods.

You can write IDL for an object class or use tools to generate appropriate IDL. For example, you can use the Java `rmic -idl` command to generate IDL files from an enterprise bean's remote and home interfaces.

If you are writing your own IDL files, or want to examine existing IDL files, consider the following reference information about IDL:

- [“IDL name scoping” on page 2](#)
- [“IDL Interface declarations” on page 3](#)
- [“Multiple IDL interfaces” on page 10](#)
- [“IDL include directives” on page 11](#)
- [“IDL pragma directives” on page 11](#)
- [“The idlc command \(IDL compiler\)” on page 13](#)
- [“The IDL-to-Java compiler” on page 17](#)
- [“Conventions used in documenting IDL syntax” on page 21](#)
- [“IDL lexical rules” on page 21](#)
- [“IDL reserved words” on page 22](#)
- [“Syntax for comments in IDL code” on page 22](#)

IDL name scoping

An IDL file forms a naming scope (referred to, in short, as a scope). Modules, interface statements, structures, unions, methods (operations), and exceptions form nested scopes. An identifier can only be defined one time in a particular scope. Identifiers can be redefined in nested scopes.

Names can be used in an unqualified form within a scope, and the name will be resolved by successively searching the enclosing scopes. When an unqualified name is defined in an enclosing scope, that name cannot be redefined.

Fully-qualified names have the following form:

```
scope-name::identifier
```

For example, the method name `mymethod` defined within the interface `Test` of module `M1` has the fully-qualified name:

```
M1::Test::mymethod
```

A qualified name is resolved by first resolving the *scope-name* to a particular scope, *S*, and then locating the definition of the *identifier* within that scope. Scopes that enclose the scope *S* are not searched.

Qualified names can also have the following form:

```
::identifier
```

These names are resolved by locating the definition of *identifier* within the outermost name scope.

Every name defined in an IDL specification is given a global name, constructed as follows:

- Before the IDL compiler scans the IDL file, the name of the current root and the name of the current scope are empty. When each module is encountered, the string of two colons (::) and the module name are appended to the name of the current root. At the end of the module, they are removed.
- When each interface, struct, union, or exception definition is encountered, the string of two colons (::) and the associated name are appended to the name of the current scope. At the end of the definition, they are removed. While parameters of an operation declaration are processed, a new unnamed scope is entered so that parameter names can duplicate other identifiers.
- The global name of an IDL definition is then the concatenation of the current root, the current scope, two colons (::), and the local name for the definition.

The names of types, constants, and exceptions defined by base interfaces are accessible in a derived interface. References to these names must be unambiguous. Ambiguities can be resolved by using a scoped name (prefacing the name with the name of the interface that defines it, and the two colons (::), as in *base-interface::identifier*). Scope names can also be used to refer to a constant, type, or an exception name defined by a base interface but redefined by a derived interface.

IDL interface declarations

The IDL specification for a class of objects must contain a declaration of the interface these objects will support.

IDL interfaces and types should be enclosed inside a module scope. IDL declared outside of a module scope takes up namespace in the global IDL namespace and risks having name collisions with names declared by other IDL developers. For more information, see [“IDL name scoping” on page 2](#).

When objects are implemented using classes, the interface name is used as a class name as well. In addition to the interface name and its base interface names, an interface indicates new methods (operations), and any constants, type definitions, and exception structures that the interface exports.

An interface declaration has the following syntax:

```
interface interface-name [: base-interface1, base-interface2, ...]
{
    constant declarations
    type declarations
    exception declarations
    attribute declarations
    operation declarations
};
```

All of the declaration elements are optional, and their order is not usually significant. However you must bear in mind the following considerations:

- Interface names must be declared before they are referenced.
- Types, constants, and exceptions, as well as interface declarations, must be defined before they are referenced (as in C or C++).
- Using one declaration can mandate another, and determine the order in which they are declared. For example, if an operation raises an exception, the exception must be declared and must come before the operation in the list.

The base-interface names specify the interfaces from which *interface-name* is derived.

Parent-interface names are required only for the immediate base interfaces. Each base interface must have its own IDL specification (which must be #included in the IDL file). A base interface cannot be named more than one time in the interface statement header.

The following topics describe the declaration elements that can be specified within the body of an interface declaration:

- [“IDL constant declarations” on page 4](#) .
- [“IDL type declarations” on page 4](#) .
- [“IDL exception declarations” on page 7](#) .
- [“IDL attribute declarations” on page 8](#) .
- [“IDL operation declarations” on page 8](#) .

IDL constant declarations

Constants are declared in IDL just as in C++, except that the type of the constant must be a valid IDL type. IDL Constant declarations take the following form:

- `const const-type identifier = constant-expression ;`

The *const-type* must be a valid IDL integer, char, boolean, floating point, string, or user-defined type name. The *identifier* is the name of the constant being defined. The *constant-expression* is a constant expression as in C or C++, and can include the usual C or C++, unary and binary operators (|, ^, &, >>, <<, +, -, *, /, %, ~), parentheses for controlling operator precedence, literal values (integer, string, character, and floating point), and the boolean literal values TRUE and FALSE.

IDL type declarations

IDL specifications can include the following type declarations as in C++, with the restrictions and extensions described in these topics:

- IDL Basic types:
 - [“IDL integer types” on page 4](#)
 - [“IDL floating point types” on page 5](#)
 - [“IDL character type” on page 5](#)
 - [“IDL boolean type” on page 5](#)
 - [“IDL octet type” on page 5](#)
 - [“IDL any type” on page 5](#)
- [“IDL constructed types” on page 5](#)
- [“IDL template types” on page 6](#)
- [“IDL arrays” on page 7](#)
- [“IDL object types” on page 7](#)

The form of a type declaration within the body of an interface declaration is described in [“IDL interface declarations” on page 3](#) .

IDL integral types

IDL supports only the following integral types, which represent the corresponding value ranges:

Table: 1. Supported IDL integer types and their value ranges

| Integral type | Value range |
|---------------|---|
| short | -2 ¹⁵ through (2 ¹⁵)-1 |

| | |
|----------------|---|
| long | -2 ³¹ through (2 ³¹)-1 |
| unsigned short | 0 through (2 ¹⁶)-1 |
| unsigned long | 0 through (2 ³²)-1 |

IDL floating point types

IDL supports the float and double floating-point types. The float type represents the IEEE single-precision floating-point numbers; double represents the IEEE double-precision floating-point numbers.

Because returning floats and doubles by value might not be compatible across all Windows compilers, client programs should return floats and doubles by reference.

IDL character type

IDL supports a char type, which represents an 8-bit quantity. The ISO Latin-1 (8859.1) character set defines the meaning and representation of graphic characters. The meaning and representation of null and formatting characters is the numerical value of the character as defined in the ASCII (ISO 646) standard. Unlike C and C++, type char cannot be qualified as signed or unsigned. (The ["octet type" on page 5](#) can be used in place of unsigned char.)

IDL boolean type

IDL supports a boolean type for data items that can take only the values zero (FALSE) and one (TRUE).

IDL octet type

IDL supports an octet type, an 8-bit quantity guaranteed not to undergo conversion when transmitted between a client and server process. The octet type can be used in place of the ["unsigned char" on page 5](#) type.

IDL any type

IDL supports an any type, which permits the specification of values of any IDL type. Conceptually, an any consists of a value and a TypeCode that represents the type of the value. The TypeCode class provides functions for obtaining information about an IDL type.

IDL constructed types

In addition to the ["basic types" on page 4](#), IDL also supports three constructed types:

- Structure (struct).
- Union (union).
- Enumeration (enum).

The *structure* and *enumeration* types are specified in IDL just as they are in C and C++, with the following restrictions:

- Recursive type specifications are allowed only through the use of the sequence template type.
- Structures and enumerations in IDL must be tagged. For example, struct { int a; ... } is an inappropriate type specification (because the tag is missing). The tag introduces a new type name.
- Structure and enumeration type definitions need not be part of a typedef statement; furthermore, if they are part of a typedef statement, the tag of the struct must differ from the type name being defined by the typedef. For example, the following are valid IDL struct and enum definitions:

```
struct myStruct {
    long x;
    double y;
};
```

```
/* defines type name myStruct */
enum colors { red, white, blue };
/* defines type name colors */
```

The following IDL definitions are **not** valid:

```
typedef struct myStruct {
    /* NOT VALID */
    long x;
    /* Tag myStruct is the same */
    double y;
    /* as the type name below; */
} myStruct;
/* myStruct has been redefined */
typedef enum colors { red, white, blue } colors;
/* NOT VALID */
```

The IDL *union* type is a cross between the C union and switch statements. This type is specified in IDL just as it is in C and C++, with the restriction that discriminated unions in IDL must be tagged. The syntax of a union type declaration is as follows:

```
union identifier switch
    (switch-type) { case+ }
```

- The *identifier* following the union keyword defines a new legal type. (Union types can also be named using a typedef declaration.)
- The *switch-type* specifies an integral, character, boolean, or enumeration type, or the name of a previously defined integral, boolean, character or enumeration type.
- Each case of the union is specified with the following syntax:

```
case-label+ type-spec declarator ;
```

Where

- Each *caselabel* has one of the following forms:

```
case const-expr :
```

default: The *const-expr* is a constant expression that must match or be automatically castable to the *switch-type*. A default case can appear no more than once.

- *type-spec* is any valid type specification.
- *declarator* is an identifier or an array declarator (such as, foo[3][5]).

Note: A deviation from CORBA specifications exists; there is no support of longlong discriminators in unions.

IDL template types

IDL defines two template types not found in C and C++: sequences and strings. A sequence is a one-dimensional array with two characteristics: an optional maximum size (specified at compile time) and a length (determined at run time). Sequences permit passing unbounded arrays between objects. Sequences are specified as follows:

- *sequence simple-type [, positive-integer-const]*

where *simple-type* specifies any valid IDL type, and the optional *positive-integer-const* is a constant expression that specifies the maximum size of the sequence (as a positive integer).

A string is similar to a sequence of type char. It can contain all possible 8-bit quantities except NULL. Strings are specified as follows:

- *string [positive-integer-const]*

where the optional *positive-integer-const* is a constant expression that specifies the

maximum size of the string (as a positive integer, which does not include the extra byte to hold a NULL as required in C or C++).

- Since CORBA gives no specific rules on how to process blanks contained within strings, IBM WebSphere Application Server treats

```
"ABC "
```

and

```
"ABC "
```

as referring to different managed objects. If you do not want blanks to be treated as significant you should pre-process your code to either remove trailing blanks, or to add trailing blanks to some fixed string length.

IDL arrays

Multidimensional, fixed-size arrays can be declared in IDL as follows:

- *identifier* { [*positive-integer-const*] }+

where the *positive-integer-const* is a constant expression that specifies the array size, in each dimension, as a positive integer. The array size is fixed at compile time.

IDL object types

The name of the interface to a class of objects can be used as a type name. For example, if an IDL specification includes an [“interface declaration” on page 3](#) for a class (of objects) C1, then C1 can be used as a type name within that IDL specification. When used as a type, an interface name indicates a reference to an object that supports that interface. An interface name can be used as the type of an operation argument, as an operation return type, or as the type of a member of a [“constructed type \(a struct, union, or enum\)” on page 5](#). In all cases, the use of an interface name indicates a reference to (instead of an instance of) an object that supports that interface.

IDL exception declarations

IDL specifications can include exception declarations, which define data structures to be returned when an exception occurs during the execution of an operation. A name is associated with each type of exception. Optionally, a struct-like data structure for holding error information can also be associated with an exception. Exceptions are declared as follows:

```
exception identifier
{
    member*
};
```

The *identifier* is the name of the exception, and each *member* has the following form:

```
type-spec declarators ;
```

The *type-spec* is a valid IDL type specification and *declarators* is a list of identifiers or array declarators, delimited by commas. The members of an exception structure should contain information to help the caller understand the nature of the error. The exception declaration can be treated like a struct definition: whatever you can access in an IDL struct, you can access in an IDL exception. Unlike a struct, an exception can be empty, meaning the exception is just identified by its name.

If an exception is returned as the outcome of an operation, the exception *identifier* indicates which exception occurred. The values of the members of the exception provide additional information specific to the exception. [“IDL operation declarations” on page 8](#) describes how to indicate that a particular operation can raise a particular exception.

The following is an example showing the declaration of a BAD_FLAG exception:

```
exception BAD_FLAG
{
```

```
}; long ErrCode; char Reason[80];
```

In addition to user-defined exceptions, there are several predefined exceptions for system run-time errors. The standard exceptions as prescribed by CORBA are subclasses of CORBA::SystemException. These exceptions correspond to standard run-time errors that can occur during the execution of any operation (regardless of the list of exceptions listed in the operation's IDL specification).

Each of the standard exceptions has the same structure: an error code (to designate the subcategory of the exception) and a completion status code. For example, the NO_MEMORY standard exception has the following definition:

```
enum completion_status
{
    COMPLETED_YES, COMPLETED_NO, COMPLETED_MAYBE
};
exception NO_MEMORY
{
    unsigned long minor;
    completion_status completed;
};
```

The "completion_status" value indicates whether the operation was never initiated (COMPLETED_NO), if the operation completed its execution prior to the exception (COMPLETED_YES), or if the operation's completion status is indeterminate (COMPLETED_MAYBE).

IDL attribute declarations

Declaring an attribute as part of an interface is equivalent to declaring one or two accessor operations: one to retrieve the value of the attribute (a get or read operation) and (unless the attribute specifies readonly) one to set the value of the attribute (a set or write operation).

Attributes are declared as follows:

```
[ readonly ] attribute type-spec declarators ;
```

where:

- *type-spec* specifies any valid IDL type (except a sequence).
- *declarators* is a list of identifiers, delimited by commas. An array declarator cannot be used directly when declaring an attribute, but the type of an attribute can be a user-defined type that is an array. Although the type of an attribute cannot be a sequence, it can be a user-defined type that is a sequence. The optional readonly keyword specifies that the value of the attribute can be accessed but not modified. (In other words, a readonly attribute has no set operation.) Below are examples of attribute declarations, which are specified within the body of an interface statement:

```
interface Goodbye: Hello
{
    void sayBye();
    attribute short xpos;
    attribute char c1, c2;
    readonly attribute float xyz;
};
```

Attributes are inherited from base interfaces. An inherited attribute name cannot be redefined to be a different type.

IDL operation declarations

Operation declarations define the interface of each operation introduced by the interface. (An IDL operation is typically implemented by a method in the implementation programming language. Hence, the terms operation and method are often used interchangeably.) An operation declaration is similar to a C++ virtual function definition:

```
[ oneway ] type-spec identifier ( parameter-list ) [ raises-expr [ context-expr ] ] ;
```

where

- *identifier* is the name of the operation.
- *type-spec* is any valid IDL type, except a sequence, or the keyword `void`, indicating that the operation returns no value. (Although the return type cannot be a sequence, it can be a user-defined type that is a sequence.) Unlike C and C++ procedures, operations that do not return a result must specify `void` as their return type.

The remaining syntax of an operation declaration is elaborated in the following topics:

- [“IDL operation declarations: “oneway” keyword” on page 9](#) .
- [“IDL operation declarations: parameter list” on page 9](#) .
- [“IDL operation declarations: “raises” expression” on page 9](#) .
- [“IDL operation declarations: “context” expression” on page 10](#).

IDL operation declarations: “oneway” keyword

For an overview of IDL operation declarations, see [“IDL operation declarations” on page 8](#) .

The optional `oneway` keyword specifies that when a caller invokes the operation, no reply is expected or received. The invocation semantics of a `oneway` operation are *best-effort*, which does not guarantee delivery of the call. *Best-effort* implies that the operation will be invoked at most once. A `oneway` operation must not have any output parameters and must have a return type of `void`. A `oneway` operation also must not include a `raises` expression.

If the `oneway` keyword is not specified, then the operation has *at-most-once* invocation semantics if an exception is raised, and it has *exactly-once* semantics if the operation succeeds. This means that an operation that raises an exception has been implemented zero or one times, and an operation that succeeds has been implemented exactly once.

IDL operation declarations: parameter list

For an overview of IDL operation declarations, see [“IDL operation declarations” on page 8](#) .

The parameter-list contains zero or more parameter declarations for the operation, delimited by commas. (The target object for the operation is not explicitly specified as an operation parameter in IDL.) If there are no explicit parameters, the syntax `()` must be used, rather than `(void)`. A parameter declaration has the following syntax:

- `{ in | out | inout } type-spec declarator`
where *type-spec* is any valid IDL type (except a sequence), and *declarator* is an identifier or an array declarator. Although the type of a parameter cannot be a sequence, it can be a user-defined type that is a sequence.

The required `in|out|inout` directional attribute indicates whether the parameter is to be passed from caller to callee (`in`), from callee to caller (`out`), or in both directions (`inout`). The following are examples of valid operation declarations:

```
short meth1(in char c, out float f);
oneway void meth2(in char c);
float meth3();
```

An operation's implementation should not modify an `in` parameter. If a change must be made by the implementation, the implementation should copy the parameter and only modify the copy.

If an operation raises an exception, the values of the return result and the values of the `out` and `inout` parameters (if any) are undefined.

IDL operation declarations: “raises” expression

For an overview of IDL operation declarations, see [“IDL operation declarations” on page 8](#) .

The optional raises expression in an IDL operation declaration indicates which exceptions the operation can raise. A raises expression is specified as follows:

```
raises ( identifier1 , identifier2 , ... )
```

where each *identifier* is the name of a previously defined exception. In addition to the exceptions listed in the raises expression, an operation can also signal any of the standard exceptions. Standard exceptions, however, should not appear in a raises expression. If no raises expression is given, then an operation can raise only the standard exceptions. [“IDL exception declarations” on page 7](#) contains further information on defining exceptions and the list of standard exceptions.

IDL operation declarations: "context" expression

For an overview of IDL operation declarations, see [“IDL operation declarations” on page 8](#) .

The optional context expression (context-expr) in an operation declaration indicates which elements of the caller's context the operation's implementation can consult. A context expression is specified as follows:

```
context ( identifier1 , identifier2 , ... )
```

where each *identifier* is a string literal made up of alphanumeric characters, periods, underscores and asterisks. The first character must be alphabetic, and an asterisk can only appear as the last character, where it serves as a wildcard matching any characters. If convenient, identifiers can consist of period-separated valid identifier names, but that form is optional.

The Context is a special object that is specified by the CORBA standard. It contains a property list: a set of property-name/string-value pairs that the caller can use to store information about its environment that operations can find useful. It is used in much the same way as environment variables. It is passed as an additional parameter to operations that are defined as context-sensitive in IDL.

The context expression of an operation declaration in IDL specifies which property names the operation uses. If these properties are present in the Context object supplied by the caller, they will be passed to the object implementation, which can access them through the interface of the Context object.

The argument that is passed to the operation having a context expression is a Context object, not the names of the properties. The caller must create a Context object and use the interface of the Context object to set the context properties. The caller then passes the Context object in the operation invocation. The CORBA standard allows properties in addition to those in the context expression to be passed in the Context object.

Multiple IDL interfaces

A single IDL file can define multiple interfaces. When a file defines two or more interfaces that reference one another, forward declarations can be used to declare the name of an interface before it is defined. This is done as follows:

```
interface interfaceName ;
```

The actual definition of the interface for *interfaceName* must appear later in the same IDL file.

If multiple interfaces are defined in the same IDL file, they can be grouped into modules, by

using the following syntax:

```
module moduleName { definition+ };
```

where each definition is a type declaration, constant declaration, exception declaration, interface statement or nested module statement. Modules are used to scope identifiers.

Alternatively, multiple interfaces can be defined in a single IDL file without using a module to group the interfaces. Whether a module is used for grouping multiple interfaces or not, the languages bindings produced from the IDL file will include support for all of the defined interfaces.

IDL include directives

If your interface declaration refers to a parent interface, or uses some other referenced types, the IDL file must contain `#include` statements that tell the IDL compiler where to find the referenced interface definitions (the IDL files).

If your interface declaration refers to IDL types (defined by the CORBA specification) that are not IDL reserved words, then the IDL file should contain an `#include` statement for the `orb.idl` file.

As in C and C++, if an `#include` statement specifies a filename that is enclosed in angle brackets (`[]`), the search for the file begins in system-specific locations. If the filename is enclosed in double quotation marks (`" "`), the search for the file begins in the current working directory, before searching the system-specific locations.

For information on other preprocessor directives that can be used in IDL, see [“IDL pragma directives” on page 11](#).

IDL pragma directives

IBM WebSphere Application Server supports the following pragmas:

- [“localonly” on page 11](#)
- [“localonly abstract” on page 12](#)
- [“cpponly” on page 12](#)
- [“init” on page 12](#)
- [“ID” on page 12](#)
- [“Prefix” on page 12](#)
- [“version” on page 13](#)

For information on other preprocessor directives that can be used in IDL, see [“IDL include directives” on page 11](#).

localonly pragma

This pragma supports the generation of bindings for objects that are known to be local (not distributed). This pragma can occur at any point in the IDL file following the definition or forward declaration of the designated interace.

The syntax is:

```
#pragma meta interface-name localonly
```

The IDL interface identified by *interface-name* is treated by generated bindings as strictly local to the caller's process. No calls to the CORBA ORB occur when invoking the

operations defined in this interface. *interface-name* can be a simple name of an interface in the current scope or a fully- or partially-qualified interface name. The interface must be previously defined or forward declared when the pragma statement is encountered.

localonly abstract pragma

This pragma is like the localonly pragma, but it signifies an abstract function that cannot be instantiated. These types of interfaces are used to just define interfaces.

The syntax is:

```
#pragma meta interface-name localonly abstract
```

cpponly pragma

This pragma suppresses the generation of IOM interlanguage bindings.

The syntax is:

```
#pragma meta interface_name cpponly
```

In the default case, without this pragma, two sets of bindings are produced:

- The standard CORBA C++ bindings suitable for use with the ORB component.
- IOM bindings suitable for interlanguage interaction.

Without this pragma, only the standard CORBA C++ bindings are produced.

init pragma

This pragma specifies a function to use to initialize newly created objects.

The syntax is:

```
#pragma meta method-name init
```

This pragma allows the IDL to specify the name of a function to be used to initialize the newly created method. When this pragma is not used, the emitters produce a `_create()` function that takes no parameters and does no initialization after the new object is created.

For example, if the IDL contains:

```
interface A
{
    void initFunction(int);
};
#pragma meta A::initFunction init
```

the C++ class A that implements interface A will have a `_create()` function that takes an int parameter (because `initFunction` takes an int). Also, the code inside `_create(int)` creates a new instance of class A and then call `initFunction(int)` on the newly created object, passing along its int parameter.

ID pragma

This CORBA-defined pragma overrides the default RepositoryID for an IDL entity.

The syntax is:

```
#pragma ID scoped-name literal-string
```

which sets the RepositoryID of *scoped-name* to *literal-string* instead of the default Repository ID.

Prefix pragma

This CORBA-defined pragma sets the RepositoryID prefix

The syntax is:

```
#pragma prefix string
```

which sets the current prefix used in generating OMG IDL format RepositoryIDs. The specified prefix applies to RepositoryIDs generated after the pragma until the end of the current scope is reached or another prefix pragma is encountered.

version pragma

This CORBA-defined pragma sets the RepositoryID version number.

The syntax is:

```
#pragma version scoped-name major .minor
```

which uses the *major.minor* as the version number for RepositoryID of the *scoped-name* .

The idlc command (IDL compiler)

The Interface Definition Language Compiler (idlc) command creates usage and implementation bindings for interfaces described in IDL files. You use this command to compile one or more files containing CORBA 3.5-compliant IDL statements, and (optionally) to emit generated language bindings appropriate to each named input file.

The syntax for the idlc command is:

```
idlc [options] filename...
```

where *options* are as described in [“Options for the idlc command” on page 14](#), and *filename* is one (or more, by use of a wildcard character) of your IDL files.

The *filename* can be specified with or without a file name extension. If no file name extension is supplied, it is assumed to be ".idl". The wildcard character or asterisk (*) is permitted to appear one time in the non-path portion of the filename. For example, the following are acceptable ways to refer to xyz.idl in directory E:\idl\src:

- E:\idl\src\xyz.idl
- E:\idl\src\xyz
- E:\idl\src*.idl (all IDL files)
- E:\idl\src\x*.idl (all IDL files starting with x)
- xyz.idl (if E:\idl\src is the current directory)
- xyz (if E:\idl\src is the current directory)
- x*(if E:\idl\src is the current directory)

Any of the idlc command options can also be specified in the environment by adding the option to the string named IDLC_OPTIONS environment variable. Options specified in the IDLC_OPTIONS variable are treated as if they were keyed on the command line before any of the actual command line options. For example, if:

```
IDLC_OPTIONS="-mcpponly -mdllname=mydll"
```

and the command line is:

```
idlc -ehh idlfile
```

the result is the same as if the IDLC_OPTIONS variable was not set and the command line

was:

```
idlc -mcpponly -mdllname=mydll -ehh idlfile
```

Emitters (see `-e emit-list` in “Options for the idlc command” on page 14) can also be specified in an emit-list held in the IDLC_EMIT environment variable. When you run the idlc command, it looks for emitters specified by the `-e` or `-s` options, and also looks in any IDLC_EMIT environment variable. If it cannot find an *emit-list* in either source, then only the syntax of the named files is checked, and any errors are reported. When a compilation error (but not a warning) is detected for a particular input file, the emit phase for that file is skipped.

When all specified input files are compiled, the idlc command returns a value of zero if no errors were detected; otherwise, a non-zero value is returned.

Options for the idlc command

Options for the idlc command are preceded with a dash (-) character and can be specified individually or run together. For example, `-p -v -V` or `-pvV` are acceptable.

Some options accept an argument. Where several options have the same argument, these options can also be specified individually or run together. For example, `-p -m tie` or `-pm tieare` acceptable.

The space between the option and its argument is optional. For example, either `-mtie` or `-m tieare` acceptable.

All options are case-sensitive, even on platforms where filenames are not case-sensitive.

The following table describes each available option:

Table: 1. idlc command options

| Option | Description |
|---------------------------------------|--|
| <code>-d directory-name</code> | Specifies the directory in which to place emitted output files and directories. If none is specified, the default is the current directory. |
| <code>-V</code> | Shows the version number of the idlc command. |
| <code>-v</code> | Specifies verbose mode. This shows all internal commands (and their arguments) issued by the idlc command. |
| <code>-?</code> (or <code>-h</code>) | Writes a brief description of the idlc command syntax to standard output. |
| <code>-D define-expression</code> | Predefines a preprocessor variable for the IDL compiler. |
| <code>-I include-directory</code> | Adds a directory to the list of directories used by the IDL compiler to find <code>#include</code> files. In addition to the <code>-I</code> option, the IDLC_INCLUDE environment variable can be used to specify a list, with <i>include-directory</i> names separated by the PATH separator character. |
| <code>-i file-name</code> | Specifies the name of a file to be compiled that does not have the <code>.idl</code> extension. The <i>file-name</i> should not have an implicit <code>.idl</code> suffix added to its name. |

| | |
|-----------------------------|--|
| -p | Used as a shorthand for -D__PRIVATE__. |
| -e (or -s) <i>emit-list</i> | <p>Specifies a list of emitters to run. Emitters generate output files that contain language-specific usage and implementation bindings appropriate to each named input file. The rules used to generate the names of these output files are described in the following topics:</p> <ul style="list-style-type: none"> • “The idlc command: Emitted C++ filenames” on page 16 <p>Each emitter in the list is separated from the others by a colon (:) or semicolon (;) character. Valid emitter names are:</p> <p>hh</p> <p>Produces C++ usage bindings. If no modifiers are present, bindings with support for remotable cross-language operation are produced. The cponly, localonly, and somthis modifiers cause specialized bindings to be produced (see <i>-mname[=value]</i>).</p> <p>sc</p> <p>Produces a C++ skeleton for the basic object adapter of the ORB. If no modifiers are present, bindings with support for remotable cross-language operation are produced. The cponly, localonly, and somthis modifiers cause specialized bindings to be produced (see <i>-mname[=value]</i>).</p> <p>uc</p> <p>Produces local implementations needed by the C++ usage bindings. If no modifiers are present, bindings with support for remotable cross-language operation are produced. The cponly, localonly, and somthis modifiers cause specialized bindings to be produced (see <i>-mname[=value]</i>).</p> <p>ih</p> <p>Produces a C++ implementation header.</p> <p>ic</p> <p>Produces a template file for the C++ managed object implementation code.</p> <p>ir</p> <p>Updates the CORBA Interface Repository with the interfaces in this compilation unit.</p> |
| -m <i>name[=value]</i> | <p>Specifies an output modifier. A modifier can be given as a name or a name=value expression. The emitters are sensitive to the following modifiers:</p> <p>LINKAGE=value</p> <p>Used to insert customized C++ linkage modifiers into the generated bindings.</p> <p>notccconsts</p> <p>Eliminates the generation of C++ TypeCode constants and overloaded any operators.</p> <p>tie</p> |

| | |
|----|---|
| | <p>Generates "tie-style" bindings that assume delegation rather than inheritance.</p> <p>cponly</p> <p>Suppresses the production of cross-language bindings and produces standard CORBA C++ bindings suitable for use with a standalone ORB. cponly affects the bindings produced by the hh, sc, and uc emitters.</p> <p>localonly</p> <p>Generates bindings that can only be used to access a local object for all of the most-derived interfaces in the IDL file.</p> <p>orbadaptor</p> <p>Generates C++ bindings that allow the C++ ORB to dispatch Java implementations.</p> <p>IRforce</p> <p>Forces the interface repository (IR) emitter to destroy objects already present in the IR with the same name as in the IDL being produced.</p> <p>dllname=<i>value</i></p> <p>Puts Windows NT import or export, or both, specifications into classes contained in the DLL named by <i>value</i>.</p> <p>preInclude=<i>file-name</i></p> <p>Adds the line:</p> <pre>#include <i>file-name</i></pre> <p>to the .hh file, just before the line that includes corba.h.</p> <p>postInclude=<i>file-name</i></p> <p>Adds the line:</p> <pre>#include <i>file-name</i></pre> <p>just before the end of the .hh file.</p> |
| -J | <p>Passes options through to the Java interpreter used internally. For example:</p> <pre>-J"-mx32m"</pre> <p>sets the heap size for the interpreter to 32M.</p> |

The idlc command: Emitted C++ filenames

The names of the generated output files are derived from the filename of the corresponding IDL file. For a file named *filestem.idl*, the following list of output files can be emitted when the idlc command is run. The list contains the emitter and its corresponding output file name.

hh

filestem.hh

sc

filestem_S.cpp

uc

filestem_C.cpp

ih
filestem.ih

ic
filestem_l.cpp

The IDL-to-Java compiler

The IDL-to-Java compiler generates Java bindings for a given IDL file.

The command to invoke the IDL-to-Java code compiler has the general form:

```
idlj [options] source_IDL
```

where *source_IDL* is the name of a file that contains IDL definitions, and [options] is any combination of the options listed in [“Options for the IDL-to-Java compiler” on page 17](#).

Options for the IDL-to-Java compiler

Options can appear in any order, but must precede the IDL file specification.

The following table describes each available option:

Table: 1. IDL-to-Java command options

| Option | Description |
|--------------------------------|--|
| -d <i>symbol</i> | Defines a symbol before compilation. This is equivalent to putting the line <code>#define symbol</code> in an IDL file. It is useful when you want to define a symbol for compilation that is not defined within the IDL file, for example to include debugging code in the bindings. |
| -emitAll | Emits all types, including those found in <code>#include</code> files. By default, only those types found in <i>idl file</i> are emitted. For more information see “Emitting bindings for include files” on page 19 . |
| -fside | Defines what bindings to emit. <i>side</i> is one of <code>client</code> , <code>server</code> , <code>all</code> , <code>serverTie</code> , and <code>allTie</code> . Assumes <code>-fclient</code> if the flag is not specified. For more information see “IDL-to-Java: Emitting client and server bindings” on page 18 |
| -i <i>include_path</i> | By default, the current directory is scanned for included files. This option adds another directory. For more information see “Specifying alternative locations for include files” on page 18 . |
| -keep | Preserves preexisting bindings. The default is to generate all files without considering if they already exist. If the Java binding files do already exist, this option stops the compiler from overwriting them. Useful if you have customized those files (which you should not do unless you are very comfortable with their contents). |
| -pkgPrefix <i>type package</i> | Wherever <i>type</i> is encountered, ensures it resides within <i>package</i> in all generated files. Note: <i>type</i> must be a fully-qualified, Java-style name. |

| | |
|--|--|
| | For more information see “Inserting package prefixes” on page 20. |
| <code>-td <i>target_directory</i></code> | By default, the compiler outputs bindings to the directory from which it was invoked (the current directory). To direct the output to another directory, specify the target directory immediately following the <code>-td</code> flag. The target directory can be absolute or relative. |
| <code>-v, -verbose</code> | Generates status messages so that you can track the progress of compilation. If this option is not selected, no messages are output unless there are errors. |
| <code>-version</code> | Displays the build version of the IDL-to-Java compiler. Any additional options appearing on the command-line are ignored. Note: Version information also appears within the bindings generated by the compiler. |

IDL-to-Java: Emitting client and server bindings

To generate the Java bindings for an IDL file named `My.idl`, set the current working directory to that containing `My.idl` and issue the following command:

```
idlj My.idl
```

This command generates client-side bindings only and is equivalent to:

```
idlj -fclient My.idl
```

Client-side bindings include all generated files except the Skeleton. If you want to generate server-side bindings for `My.idl`, issue the command:

```
idlj -fserver My.idl
```

This command generates all client-side bindings plus an inheritance-model Skeleton (`ImplBase`). Currently, server-side bindings include all generated files, even the Stub. Thus, the command above is currently equivalent to each shown below:

```
idlj -fclient -fserver My.idl
idlj -fall My.idl
```

The compiler generates inheritance-model Skeletons by default. Given an interface `My` defined in `My.idl`, the compiler generates Skeleton `_MyImplBase.java`. You provide the implementation for `My`, which must extend `_MyImplBase`.

IDL-to-Java: Specifying alternative locations for include files

If `My.idl` included another IDL file, `MyOther.idl`, the compiler assumes that `MyOther.idl` resides in the local directory. If it resides in directory */includes*, for example, you would invoke the compiler with the following command:

```
idlj -i /includes My.idl
```

If `My.idl` also included `Another.idl` that resided in */moreIncludes*, then you would invoke the compiler as:

```
idlj -i /includes -i /moreIncludes My.idl
```

You can begin to see that if you have a number of places where included files can come from, the command will become long and unmanageable. So there is another means of indicating to the compiler where to search for included files. This technique is very similar to the idea of an environment variable. You must create a file called `idl.config` in a directory

that is listed in your CLASSPATH. Inside of idl.config you must provide a line of the following form:

```
includes=/includes;/moreIncludes
```

The compiler take the first version of the file it locates and read in its includes list. Notice that in this example, the separator character between the two directories is a semicolon (;). This separator character is platform-dependent. The separator character is a semicolon (;) on Windows NT, and it is a colon (:) on AIX.

Note: Some platforms will fail when issuing a long command line. If the command line to invoke the compiler becomes too long, use the idl.config file.

IDL-to-Java: Emitting bindings for include files

By default, only those interfaces, structs, and so on, that are defined in the IDL file on the command line have the Java bindings generated for them. The types defined in included files are not generated. For example, assume the following two IDL files:

```
My.idl
#include MyOther.idl
interface My
{
};
MyOther.idl
interface MyOther
{
};
```

The following command will only generate bindings for types within My:

```
idlj My.idl
```

To generate bindings for all of the types in My.idl and all of the types in files that My.idl includes (in this example, MyOther.idl), use the following command:

```
idlj -emitAll My.idl
```

There is a caveat to the default rule. #include statements which appear at the global scope are treated as described. These #include statements can be thought of as import statements. #include statements which appear within some enclosing scope are treated as true #include statements, meaning that the code within the included file is treated as if it appeared in the original file and, therefore, Java bindings are emitted for it. Here is an example:

```
My.idl
#include MyOther.idl
interface My
{
  #include Embedded.idl
};
MyOther.idl
interface MyOther
{
};
Embedded.idl
enum E {one, two, three};
```

Running the following command:

```
idlj My.idl
```

will generate the following list of Java files:

```
./MyHolder.java
./MyHelper.java
./_MyStub.java
./MyPackage
./MyPackage/EHolder.java
./MyPackage/EHelper.java
./MyPackage/E.java
./My.java
```

Notice that MyOther.java was not generated because it is defined in an import-like #include. But E.java was generated because it was defined in a true #include. Notice also that

because Embedded.idl was included within the scope of the interface My it appears within the scope of My (that is, in MyPackage).

If the -emitAll flag were used in the previous example, all types in all included files would be emitted.

IDL-to-Java: Inserting package prefixes

This option has the following form:

```
-pkgPrefix type package
```

It ensures that wherever *type* is encountered it resides within *package* in all generated files.

For example, let us suppose that a company called ABC has constructed the following IDL file:

```
Widgets.idl
module Widgets
{
  interface W1 {...};
  interface W2 {...};
};
```

Running this file through the IDL-to-Java compiler places the Java bindings for W1 and W2 within the package Widgets. But what if there is an industry convention that states that a company's packages should reside within a package named *com.company name*? Then the Widgets package does not conform. To follow the convention, it should be *com.abc.Widgets*. To place this package prefix onto the Widgets module, you implement the following:

```
idlj -pkgPrefix Widgets com.abc Widgets.idl
```

You should be aware that, if you have an IDL file which includes Widgets.idl, the -pkgPrefix flag must appear on that command as well. If it does not, then your IDL file will be looking for a Widgets package rather than a com.abc.Widgets package.

If you have a number of these packages that require prefixes, it might be easier to place them into the idl.config file as described in [“Specifying alternative locations for include files” on page 18](#). Each package prefix line should be of the form:

```
PkgPrefix.type=prefix
```

So the line for the above example would be:

```
PkgPrefix.Widgets=com.abc
```

IDL-to-Java: Emitting makefiles and specifying the path separator character

When the Java bindings are compiled using a makefile, it can become tedious to build the makefile by hand. There are two arguments to the IDL-to-Java compiler which help to build the makefile.

```
idlj -m My.idl
```

Besides the usual bindings, this will generate bfile My.u that will contain the following lines:

```
MyHelper.java: My.idl
My.java: My.idl
MyHolder.java: My.idl
MyPackage/E.java: Embedded.idl
MyPackage/EHelper.java: Embedded.idl
MyPackage/EHolder.java: Embedded.idl
_MyStub.java: My.idl
MyHelper.java \
My.java \
MyHolder.java \
MyPackage/E.java \
MyPackage/EHelper.java \
```

```
MyPackage/EHolder.java \  
_MyStub.java
```

If you are building a makefile that will run on multiple platforms, the slash (/) character is not necessarily the file separator character. Perhaps the build environment has a special variable for the file separator character. If this variable were \$(Sep), then the compiler can place this in place of the slash in My.u with the following command:

```
idlj -m -sep \$(Sep) My.idl
```

Now My.u contains the following:

```
MyHelper.java: My.idl  
My.java: My.idl  
MyHolder.java: My.idl  
MyPackage$(Sep)E.java: Embedded.idl  
MyPackage$(Sep)EHelper.java: Embedded.idl  
MyPackage$(Sep)EHolder.java: Embedded.idl  
_MyStub.java: My.idl  
MyHelper.java \  
My.java \  
MyHolder.java \  
MyPackage$(Sep)E.java \  
MyPackage$(Sep)EHelper.java \  
MyPackage$(Sep)EHolder.java \  
_MyStub.java
```

Conventions used in documenting IDL syntax

The following conventions are used in these topics to describe the syntax of IDL as specified by the CORBA standard:

| | |
|-----------------|--|
| bold | Indicates literals (such as keywords). |
| <i>variable</i> | Indicates user-supplied elements. |
| { } | Groups related items together as a single item. |
| [] | Encloses an optional item. |
| * | Indicates zero or more repetitions of the preceding item. |
| + | Indicates one or more repetitions of the preceding item. |
| | Separates alternatives. |
| – | Within a set of alternatives, an underscore indicates the default, if defined. |

IDL lexical rules

IDL generally follows the same lexical rules as C and C++. Exceptions to C++ lexical rules include:

- IDL uses the ISO Latin-1 (8859.1) character set.
- White space is ignored except as token delimiters.
- C and C++ comment styles are supported.
- IDL supports standard C or C++ preprocessing, including macro substitution, conditional compilation, and source file inclusion.
- Identifiers (user-defined names for operations, attributes, instance variables, and so on) are composed of alphanumeric and underscore characters (with the first character alphabetic) and can be of arbitrary length, up to an operating-system limit of about 250 characters.
- Identifiers must be spelled consistently with respect to case throughout a specification.
- Identifiers that differ only in case yield a compilation error.

- Within a particular name scope, there is a single namespace for all identifiers, regardless of their type. For example, using the same identifier for a constant and an interface name within the same name scope yields a compilation error.
- Integer, floating point, character, and string literals are defined as in C and C++.

IDL reserved words

The terms listed below are reserved words and cannot be used otherwise. Reserved words must be spelled using upper- and lower-case characters exactly as shown in the table. For example, "void" is correct, but "Void" yields a compilation error.

Table: 1. Reserved words for IDL

| | | | | |
|-----------|-----------|-----------|----------|-----------|
| any | default | FALSE | oneway | read-only |
| attribute | double | float | out | sequence |
| boolean | enum | in | raises | short |
| case | exception | inout | unsigned | string |
| char | *** | interface | union | struct |
| const | *** | long | void | switch |
| context | *** | module | *** | TRUE |
| *** | *** | Object | *** | *** |
| *** | *** | octet | *** | typedef |

Syntax for comments in IDL code

IDL supports both C and C++ comment styles. Two slashes (//) start a line comment, which finishes at the end of the current line. A slash and an asterisk (/*) start a block comment that finishes with an asterisk and a slash (*). Block comments do not nest. The two comment styles can be used interchangeably.

Because comments appearing in an IDL specification can be transferred to the files that the IDL Compiler generates, and because these files are often used as input to a programming language compiler, avoid using characters that are not generally allowed in comments of most programming languages. For example, the C language does not allow an asterisk and a slash (*) to occur within a comment, so its use is to be avoided, even when using C++ style comments in the IDL file.

IDL also supports throw-away comments. They can appear anywhere in an IDL specification. Throw-away comments start with the string of two slashes and a number sign (//#), and end at the end of the line. Use throw-away comments to comment out portions of an IDL specification.

The implementation registration utility (regimpl)

Before an implementation (a server program and class libraries) can be used by client applications, it must be registered in the Implementation Repository by running the implementation registration utility, regimpl.

The regimpl command can be entered at a command prompt. For WebSphere Application Server enterprise services, regimpl takes the following usage syntax, with the parameters described in the following table.

To enter interactive mode

```
regimpl
```

To add an implementation

```
regimpl -A -i alias_string [-p svr_string] [-m {on|off}] [-t  
string]
```

To update an implementation

```
regimpl -A -i alias_string [-p svr_string] [-m {on|off}] [-t  
prot_string]
```

To delete one or more implementations

```
regimpl -D -i alias_string [-i ...]
```

To list all, or selected, implementations

```
regimpl -L [-i alias_string [-i ...]]
```

To list all implementation aliases

```
regimpl -S
```

Table: 1. regimpl command parameters used by WebSphere enterprise services

| Parameter | Description |
|------------------------|--|
| -i <i>alias_string</i> | Implementation alias name (maximum of 16 -i names) |
| -p <i>svr_string</i> | Server program name (default: null) |
| -m {on off} | Enable multi-threaded server (optional) |
| -t <i>prot_string</i> | Protocol name (default: SOMD_TCPIP) |

CORBA programming

CORBA 2.1 specifies standard forms by which client code can manipulate data whose types are described using IDL. Reference material on programming these standard forms is grouped under the following topics:

- [“CORBA C++ bindings” on page 24.](#)
- [“Commonly used CORBA interfaces” on page 40.](#)
- [“C++ bindings for CORBA interfaces” on page 41.](#)
- [“CORBA programming: Storage management and _var types” on page 44.](#)

CORBA C++ bindings

C++ bindings are generated, based on CORBA 2.1 standard forms, to enable client C++ code to manipulate data whose types are described using IDL. C++ bindings that support the standard forms are called *compliant* and client code that uses (only) these forms is called *conformant*.

For more information about C++ bindings, see the following topics:

- [“CORBA C++ bindings for constants” on page 24.](#)
- [“CORBA types and business objects” on page 24.](#)
- [“CORBA C++ bindings for data types” on page 26.](#)
- [“CORBA C++ binding restrictions” on page 39.](#)
- [“CORBA programming: Name scoping and modules in the C++ bindings” on page 39.](#)

CORBA C++ bindings for constants

Constants can be defined within the IDL in either of the following ways:

- Within a module or interface.
- Globally, outside any module or interface.

If you declare an IDL constant within a module or interface, the constant is mapped as a static data item local to the C++ class for that module or interface. If you declare an IDL constant globally, the constant is mapped as a static data item global to that client application.

For example, consider the following IDL:

```
module M
{
    const string name = "testing";
};
```

After compiling the client bindings a C++ client application can refer to the constant using the expression `M::name`.

If the same constant is declared globally, outside any module or interface, then (after compiling the client bindings) a C++ client application can refer to the constant using the expression `name`.

CORBA types and business objects

Most of the CORBA types map directly onto C++ types and can be used transparently to C++. For more information on these types, see the topic [“CORBA basic types” on page 25.](#)

Other CORBA types are more complex to use because they return object references to the

caller. For more information on these types, see the topic [“CORBA types that return object references” on page 25](#).

CORBA basic types

The following basic C++ types map directly into CORBA types:

- Atomic data types:
 - Boolean
 - Char
 - Double
 - Float
 - Long
 - Octet (hexadecimal)
 - Short
 - ULONG (unsigned long)
 - UShort (unsigned short)
- Enum (enumerations)
- LongLong (long long)
- Struct
- ULONGLong (unsigned long long)
- WChar (wide character)

All these types are scoped to the class CORBA and must be declared accordingly. Their use in C++ is transparent and straightforward. For example:

```
CORBA::Short aShortvariable;  
...  
aShortVariable = 12;  
...
```

CORBA types that return object references

The following CORBA types return object references to the caller:

- Any
- Array
- Sequence
- String
- Union
- WString (wide string)

It is the responsibility of the caller to manage the object references and their associated memory. There are two facilities provided by CORBA to do this:

A_var

This is the facility most frequently used by client code because it is a smart pointer and automatically releases its object reference when it is deallocated or assigned a new object reference. This is the safest and most straightforward approach to managing these types.

A_ptr

This is a pointer type and provides the most basic object reference, which has similar semantics to a standard C++ pointer.

Note: Avoid declaring C++ Static variables as `_var`. The `_var` holds a reference to an object.

During the end of the process, this object could reference another object that was removed before end processing completes for this Static type. As a result, the `_var` could reference an inappropriate address or null pointer and thereby cause an undesirable ending.

CORBA C++ bindings for data types

C++ bindings can be created for the following CORBA data types:

- ["Any type" on page 26](#)
- ["Array types" on page 30](#)
- ["Atomic data types" on page 31](#)
- ["Enumerations" on page 32](#)
- ["Sequence types" on page 32](#)
- ["Strings" on page 35](#)
- ["Struct types" on page 36](#)
- ["Union types" on page 37](#)
- ["WStrings" on page 38](#)

C++ bindings for CORBA Any type

The purpose of the IDL "any" type is to encapsulate data of some arbitrary IDL type. The C++ bindings provide a C++ class named `CORBA::Any` that provides this functionality. A `CORBA::Any` object encapsulates a `void*` pointer and a `CORBA::TypeCode` object that describes the thing pointed to by the `void*`.

The Any type can be used with many of the CORBA types and is useful where different types can be used that are unknown to the receiver of the data, or as a common storage mechanism for passing a variety of types. It is used easily with many of the CORBA types but has a unique method of redirection operators for setting and retrieving data.

The following types are handled in this manner:

- Double
- Enumerations
- Float
- Long
- Short
- ULong
- UShort
- Unbounded Strings
- Object References

For example:

```
::CORBA::Any anything;  
anything <<= (():CORBA::Long) 123456;  
::CORBA::Long anythingStart = 123456;  
::CORBA::Long anythingLongResult = 0;  
policyVar->anything(anything);  
::CORBA::Any_var anythingResult_var(policyVar->anything());  
::CORBA::Any anythingResult(anythingResult_var);  
anythingResult >>= anythingLongResult;  
if ( anythingStart != anythingLongResult )  
{  
    cout << "Anything not set" << endl;  
    return 1;  
}  
else  
{  
    cout << "Anything set correctly..." << endl;  
}
```

There are also specialized structures provided for the following types for conversion with Any:

- Boolean
- Char
- Octet
- String

The data in an Any object is initialized and accessed using insertion (<<=) and extraction (>>=) operators defined by the C++ bindings. These operators are provided (using overloading) by CORBA::Any for each primitive data type, and are provided by the generated C++ bindings for each user-defined IDL type. As a result, there is usually no need to indicate a typecode when inserting or extracting data from a CORBA::Any (although the CORBA::Any class does provide methods for manipulate the data using an explicit TypeCode).

Types that cannot be distinguished by C++ overloading are inserted into and extracted from Any's using special *wrapper* classes. These wrapper classes are not transparent to the application; the application must explicit create and use them when inserting or extracting ambiguous types into or from Any's. For primitive IDL types that do not map to distinct C++ types (boolean, octet, and char), the wrapper classes are defined within the CORBA::Any scope; they are called from_boolean, to_boolean, from_octet, to_octet, from_char, and to_char. For information on the scope, see ["IDL name scoping" on page 2](#) . Because bounded strings cannot be distinguished in C++ from unbounded strings, CORBA::Any provides the from_string and to_string wrapper classes, for inserting/extracting bounded strings. For extracting object references from Any's as the base CORBA::Object type, CORBA::Any provides a to_object wrapper class.

For application-specific arrays, the bindings provide a special *forany* class, for inserting or extracting the array into or from an Any. For example, here is an IDL array definition:

```
typedef long LongArray[4][5];
```

For this array definition, the emitted bindings define the following:

```
typedef CORBA::Long LongArray[4][5];
typedef CORBA::Long LongArray_slice[5];
typedef LongArray_slice* LongArray_slice_vPtr;
typedef const LongArray_slice* LongArray_slice_cvPtr;
class LongArray_forany
{
public:
    LongArray_forany();
    LongArray_forany(LongArray_slice*, CORBA::Boolean nocopy=0);
    LongArray_forany(const LongArray_forany );
    LongArray_forany operator= (LongArray_slice*);
    LongArray_forany operator= (const LongArray_forany );
    ~LongArray_forany();
    const LongArray_slice operator[] (int) const;
    const LongArray_slice operator[] (CORBA::ULong) const;
    LongArray_slice operator[] (int);
    LongArray_slice operator[] (CORBA::ULong);
    operator LongArray_slice_cvPtr () const;
    operator LongArray_slice_vPtr ();
};
void operator<<=(Any , const LongArray_forany );
CORBA::Boolean operator>>=(Any , LongArray_forany );
```

Note: The nocopy optional parameter of the *_forany*'s second constructor indicates whether the *_forany* makes a copy of the input array or assumes ownership of it. (The default is for the *_forany* to assume ownership of the input array; that ownership will then be transferred to the Any when the *_forany* is inserted into the Any.)

To determine what kind of data is in Any, invoke the type method on a CORBA::Any to access a TypeCode that describes the data it holds. Alternatively, you can try to extract data of a particular type from the Any; the extraction operator returns a boolean to indicate success. If the extraction operation fails, the Any does not hold data of the type you tried to extract.

A CORBA::Any object always owns the data that its void* points to, and deletes (or releases) it when the Any is given a new value or deleted. The only question is whether this data is a copy of the data that was inserted into the Any. When primitives (including strings and enums) are inserted, a copy is made, and a copy is returned when the data is extracted.

For non-primitive (constructed) data, extraction from an Any always updates a pointer (owned by the caller) so that it points to the data owned by the Any. The caller should not, therefore, free this data or reference it after the Any has been given a new value or deleted. For constructed IDL type T, the emitted bindings define the following extraction operator:

```
CORBA::Boolean operator>>=(Any&, T*&);
```

When a reference to constructed data is inserted into an Any (when the C++ syntax looks as if you are inserting a value instead of a pointer) a copy is made. In this case, the caller retains ownership of the original data. For example, for constructed type T and interface I, the emitted bindings define the following insertion operators, which copy (or, in the case of object references, _duplicate) the inserted value):

```
void operator<<=(Any&, const T&);  
void operator<<=(Any&, T_ptr);
```

When a pointer to constructed data is inserted into an Any, as when using the following insertion operators emitted for type T and interface I:

```
void operator<<=(Any&, T*);  
void operator<<=(Any&, T_ptr*);
```

The Any takes ownership of the constructed type's top-level storage only; however, the Any makes no copy of the top-level storage or any embedded storage. All further use of the pointer that was inserted is forbidden; the Any now owns it and is free to delete it at any time. The next time data is inserted into the Any, or when the Any is destroyed, the Any deletes the previously-inserted pointer. However, if the constructed type consists of multiple dynamically-allocated regions of memory, only the top-level storage is deleted. (The Any deletes arrays using a single array delete; other constructed types are deleted using a single, normal delete.) Further, the top-level storage is deleted as a void*, rather than its true type, which means that the constructed type's destructor will not be run. Due to these restrictions, insertion by pointer of constructed types into an Any should be used with caution.

In summary, when extracting data from an Any, the caller does own the data for primitive types, but does not own the data for constructed types. When inserting data into an Any, the caller retains ownership of the data for primitive types, for constructed types inserted by value, and for storage embedded within constructed types inserted by pointer. The caller does not retain ownership of the top-level contiguous storage for a constructed type inserted into an Any by pointer.

The following is an example that illustrates the previously discussed aspects of CORBA::Any usage. The IDL for this example appears immediately below. It defines a struct and an array that will be inserted into an Any.

```
Module M  
{  
  Struct S  
  {  
    string str;  
    longlng;  
  };  
  typedef long longl[2][3];  
}
```

A C++ program illustrating Any insertion and extraction appears below:

```
#include <stdio.h>  
#include any_C.cpp  
main()
```

```

{
CORBA::Any a; // the Any that we'll be using
// test a long
long l = 42;
a <<= l;
if (a.type()->equal(CORBA::_tc_long))
{
    long v;
    a >>= v;
    printf("the any holds a long = %d\n", v);
}
else
printf("failure: long insertion\n");
// test a string
char *str = "abc";
a <<= str;
if (a.type()->equal(CORBA::_tc_string))
{
    char *ch;
    a >>= ch;
    printf("the any holds the string = %s\n", ch);
    delete ch;
    a >>= ch;
    printf(" the any still holds the string = %s\n", ch);
    delete ch;
}
else
printf("failure: string insertion\n");
// test a bounded string -- note you do not use a typecode here
char *bstr = "abcd";
char *rstr;
a <<= CORBA::Any::from_string(bstr, 6);
if (a >>= CORBA::Any::to_string(rstr,6))
printf("the any holds a Bounded string<6> = %s\n", rstr);
else
printf("failure: bounded string insertion\n");
// test a user-defined struct
M::S *s1 = new M::S;
char *saveforlater = CORBA::string_dup("abc");
s1->str = saveforlater;
s1->lng = 42;
a <<= s1; // insertion by pointer
if (a.type()->equal(_tc_M_S))
{
    M::S *sp;
    a >>= sp;
    printf("the any holds an M::S = {%s, %d}\n", sp->str, sp->lng);
}
else
printf("failure: struct insertion by pointer\n");
M::S s2;
s2.str = CORBA::string_dup("def");
s2.lng = 23;
a <<= s2; // note: this deletes *s1, but not saveforlater
printf("saveforlater still = %s\n", saveforlater);
CORBA::string_free(saveforlater);
if (a.type()->equal(_tc_M_S))
{
    M::S *sp;
    a >>= sp;
    printf("the any holds an M::S = {%s, %d}\n", sp->str, sp->lng);
}
else
printf("failure: struct insertion by value\n");
M::S_var s3 = new M::S;
s3->str = CORBA::string_dup("ghi");
s3->lng = 96;
a <<= *s3;
if (a.type()->equal(_tc_M_S))
{
    M::S *sp;
    a >>= sp;
    printf("the any holds an M::S = {%s, %d}\n", sp->str, sp->lng);
}
else
printf("failure: struct insertion by ref to value\n");
// test an array
M::long1_var llv = M::long1_alloc();
for (i=0;i<2;i++)
for (j=0;j<3;j++)
llv[i][j] = (i+1)*(j+1);
a <<= M::long1_forany(llv);
if (a.type()->equal(_tc_M_long1))
{
    M::long1_forany lls;
    a >>= lls;
    printf("the any holds the array: ");
    for (i=0;i<2;i++)
    for (j=0;j<3;j++)
    printf("%d ", lls[i][j]);
    printf("\n");
}
else printf("failure: array insertion\n");
}

```

Output from the above program is:

```

the any holds a long = 42
the any holds a string = abc
the any still holds a string = abc
the any holds a bounded string<6> = abcd
the any holds an M::S = {abc, 42}
saveforlater still = abc
the any holds an M::S = {def, 23}
the any holds an M::S = {ghi, 96}

```

```
the any holds the array: 1 2 3 2 4 6
```

C++ bindings for CORBA Array types

An IDL array type is mapped to the corresponding C++ array definition. There is also a corresponding `_var` type. For example, given the following IDL definition:

```
typedef long LongArray [4][5];
```

The C++ bindings provide the following definitions:

```
typedef CORBA::Long LongArray[4][5];
typedef CORBA::Long LongArray_slice[5];
typedef LongArray_slice* LongArray_slice_vPtr;
typedef const LongArray_slice* LongArray_slice_cvPtr;
class LongArray_var
{
public:
    LongArray_var();
    LongArray_var(LongArray_slice*);
    LongArray_var(const LongArray_var&);
    LongArray_var & operator= (LongArray_slice*);
    LongArray_var & operator= (const LongArray_var &);
    ~LongArray_var();
    const LongArray_slice& operator[] (int) const;
    const LongArray_slice& operator[] (CORBA::ULong) const;
    LongArray_slice & operator[] (int);
    LongArray_slice & operator[] (CORBA::ULong);
    operator LongArray_slice_cvPtr () const;
    operator LongArray_slice_vPtr& ();
};
LongArray_slice * LongArray_alloc();
void LongArray_free (LongArray_slice*);
LongArray_slice * LongArray_dup (const LongArray_slice*);
```

As shown above, array mappings provide `alloc`, `dup`, and `free` functions (for allocating, copying, and freeing array storage), as static member functions of the class within which the array type name is scoped. The `alloc` function dynamically allocates an array, which can be later freed using the `free` function. The `dup` function dynamically allocates an array and copies the elements of an existing array into it. A `NULL` pointer can be passed to the `free` function. None of these functions throws exceptions.

The type of the pointer returned from `LongArray_alloc` is `LongArray_slice*`. The C++ bindings define array "slice" types for all arrays declared in IDL. The reason is that using the name `LongArray` in a program does not denote the array `LongArray`; rather, it denotes a pointer to the array. For historical reasons (related to the fact that arrays are not an actual data type in C and C++) the type of this pointer has one less array dimension than the array `LongArray`. Thus, the bindings for `LongArray` include the following typedef:

```
typedef string LongArray_slice[5];
```

Hence, `LongArray_slice*` is the correct type for a pointer to an array of IDL type `LongArray`.

As with structs and sequences, arrays use special auxiliary classes for automatic storage management of `String` and object reference elements. The auxiliary classes for `Strings` and object references manage the storage just as the associated `_var` classes do.

When assigning a value to an array element that is an object reference, the assignment operator will automatically release the previous value (if any). When assigning an object reference pointer to an array element, the array assumes ownership of the pointer (no `_duplicate` is done), and the application should no longer access the pointer directly (if this is not the desired behavior, then the caller can explicitly `_duplicate` the object reference before assigning it.) However, when assigning to an object reference array element from a `_var` object or from another struct, union, array, or sequence member (rather than from an object reference pointer), a `_duplicate` is done automatically.

For an array of `Strings`, when assigning a value to an element or deleting the array, any previously held (non-null) `char*` is automatically freed. As when assigning to `String_vars`, assigning a `char*` to a string element does not make a copy, but assigning a `const char *` or another struct/union/array/sequence `String` element does make a copy. One should never

assign a string literal (for example, "abc") to a String array element without an explicit cast to "const char*". When assigning a char* that occupies static storage (rather than one that was dynamically allocated) the caller can use CORBA::string_dup to duplicate the string before assigning it.

The following is an example that involves multidimensional arrays, and array_vars, from the IDL snippet immediately below:

```
typedef string s2_3[2][3];
typedef string s3_2[3][2];
```

The code that exercises the C++ arrays that correspond to the above IDL is shown below. Notice that in the following example:

- There is no need to explicitly use slice types when working with the array _var types, because the bindings declare the pointer held by an array _var type using the appropriate slice type.
- At the end, the program explicitly frees the storage pointed to by s2_3p (using an array delete operator), but does not do this for s3_2v, because its pointer is deleted when the destructor for s3_2v is implemented. (This is the purpose of the _var types.)

```
#include arr_C.cpp
#include stdio.h
main()
{
    int i,j;
    char id[40];
    // create arrays
    s2_3_slice* s2_3p = s2_3_alloc();
    s3_2_var s3_2v = s3_2_alloc();
    // load the arrays
    for(i=0; i<2; i++)
    {
        for (j=0; j<3; j++)
        {
            sprintf(id, "s2_3 element [%d][%d]",i,j);
            // Use string_dup when assigning a char*
            // if you do not want the array to own the original:
            s2_3p[i][j] = CORBA::string_dup(id);
        }
    }
    for(i=0; i<3; i++)
    {
        for (j=0; j<2; j++)
        {
            sprintf(id, "s3_2_var element [%d][%d]",i,j);
            // Use string_dup when assigning a char*
            // if you do not want the array to own the original:
            s3_2v[i][j] = CORBA::string_dup(id);
        }
    }
    // print the array contents
    for(i=0; i<2; i++)
    {
        for (j=0; j<3; j++)
        {
            printf("%s\n", s2_3p[i][j]);
        }
    }
    for(i=0; i<3; i++)
    {
        for (j=0; j<2; j++)
        {
            printf("%s\n", s3_2v[i][j]);
        }
    }
    delete [] s2_3; // needed to prevent a storage leak.
    // Nothing is needed for s3_2v, because
    // it is a _var type.
}
```

Output from the above program is:

```
s2_3 element [0][0]
s2_3 element [0][1]
s2_3 element [0][2]
s2_3 element [1][0]
s2_3 element [1][1]
s2_3 element [1][2]
s3_2_var element [0][0]
s3_2_var element [0][1]
s3_2_var element [1][0]
s3_2_var element [1][1]
s3_2_var element [2][0]
s3_2_var element [2][1]
```

C++ bindings for CORBA Atomic data types

The atomic IDL data types (long, short, unsigned long, unsigned short, float, double, char, boolean, and octet) are mapped into types defined in `corba.h`, nested within the CORBA scope. See ["IDL name scoping" on page 2](#) for more information. The first letter of the mapped type is capitalized. For example, to introduce and initialize a local variable named `Myvar` whose type corresponds to the IDL type named `long`, a C++ programmer could employ the following expression:

```
CORBA::Long Myvar = 1;
```

The mapping for the IDL boolean type (`CORBA::Boolean`) defines only the values 0 and 1. The unsigned long and unsigned short IDL types are mapped to `CORBA::ULong` and `CORBA::UShort`, respectively.

C++ bindings for CORBA Enumerations

An IDL enum is mapped to a corresponding C++ enum. For example, given the following IDL:

```
module M
{
    enum Color
    {
        red, green, blue
    };
};
```

A C++ programmer could introduce a local variable of the corresponding C++ type and initialize it with the following code:

```
{
    M::Color MYCOLOR = M::red;
}
```

The enumeration constant `red` is not denoted using the expression `M::Color::red`. For this reason, names of enumeration constants must be carefully chosen.

C++ bindings for CORBA Sequence types

An IDL sequence type is mapped to a C++ class that behaves like an array with a current length (how many elements have been stored) and a maximum length (how much storage is currently allocated). The array indexing operator `[]` is used to read and write sequence elements. (Indexing begins at zero.) It is the programmer's responsibility to check the current sequence length or maximum to prevent accessing the sequence beyond its bounds. The length and maximum of the sequence are not automatically increased to accommodate new elements; the programmer must explicitly increase them.

The maximum length of a bounded sequence is implicit in the sequence class's type and cannot be changed. The initial maximum length of an unbounded sequence is set to zero by the default constructor, or initialized by the programmer using a non-default constructor. Setting the initial maximum length of an unbounded sequence using the non-default constructor causes storage to be allocated for the specified number of sequence elements.

Sequence classes provide an overloaded member function `length` that either returns or sets the length of the sequence. Setting the length of an unbounded sequence to a value larger than the current maximum causes the sequence to allocate new storage of the required size, copy any previous sequence elements to the new storage, free the old storage (if any), and reset the maximum to the new length. Sequence classes also provide `allocbuf` and `freebuf` member functions for explicitly allocating/freeing the sequence's storage buffer. Decreasing a sequence's length does not cause any storage to be deallocated, but any orphaned sequence elements are no longer accessible, even if the sequence length is subsequently increased.

Sequences may or may not manage (own) the storage that contains their elements, and the elements themselves. By default, a sequence manages this storage, but a *release*

constructor parameter allows client programmers to request otherwise (when passing in a buffer explicitly allocated using the `alloca` function).

The following IDL:

```
typedef sequence s1; // unbounded sequence
```

is mapped to the following C++ sequence class:

```
class s1
{
public:
    s1(); // default constructor
    s1(CORBA::ULong max); // "max" constructor
    s1(CORBA::ULong max, CORBA::ULong length,
        T* data, CORBA::Boolean release=0);
        // "data" constructor
    s1(const s1&); // copy constructor
    s1 &operator=(const s1&); // assignment operator
    ~s1(); // destructor
    CORBA::ULong maximum() const;
    CORBA::ULong length() const;
    void length(CORBA::ULong len);
    T& operator[] (CORBA::ULong index);
    const T& operator[] (CORBA::ULong index) const;
    static T* alloca(CORBA::ULong nelems);
    static void freebuf(T* data);
};
```

The default constructor sets the length and maximum to zero. (For a bounded sequence, the default constructor sets the maximum to the sequence bounds and allocates storage for the maximum number of elements, which the sequence owns.)

The "max" constructor sets the initial sequence maximum and allocates a storage buffer for the specified number of sequence elements, which the sequence owns. The length of the sequence is initialized to zero.

Note: This method is not available for bounded sequences.

The "data" constructor sets the initial length and maximum of the sequence, as well as its initial contents. (For bounded sequences, the maximum cannot be set by the "data" constructor.) The input storage should match the specified sequence maximum. Ownership of the input storage is indicated by the "release" parameter. Passing `release=1` specifies that the storage was allocated using `s1::alloca`, that the sequence should delete the storage and the sequence elements when the sequence is deleted or when the storage needs to be reallocated, and that the caller will not directly access the storage after the call (because the sequence is free to delete it at any time). In general, sequences constructed with `release=0` should not be passed as input parameters, because the callee must assume that the sequence owns the sequence elements.

The copy constructor creates a new sequence with the same maximum and length as the input sequence and copies the sequence elements to storage that the sequence owns. The assignment operator performs a deep copy, releasing the previous sequence elements if necessary. It behaves as if the destructor were run, followed by the copy constructor.

The destructor destroys each of the sequence elements (from zero through `length-1`), if the sequence owns the storage.

The `alloca` function allocates enough storage for the specified number of sequence elements; the return value can then be passed to the "data" constructor. Each sequence element is initialized using its default constructor; string elements are initialized to `NULL`; object reference elements are initialized to nil object references. `NULL` is returned if storage cannot be allocated for any reason. If ownership of the allocated buffer is not transferred to a sequence using the "data" constructor with `release=1`, the buffer should be subsequently freed using the `freebuf` function. The `freebuf` function insures that each sequence element's destructor is run (or, for strings, that `CORBA::string_free` is called, or for object references, that `CORBA::release` is called) before the buffer is deleted. The `freebuf` function ignores

NULL pointers passed to it. Neither `alloca` nor `freebuf` throw CORBA exceptions.

As with structs, sequences that manage their elements use special auxiliary classes for automatic storage management of String and object reference sequence elements. These auxiliary classes manage Strings and object references just as the associated `_var` classes do.

For a storage-managing sequence whose elements are object references, when assigning a value to an element, the assignment operator will automatically release the previous value, if any. When assigning an object reference pointer to such a sequence element, the sequence assumes ownership of the pointer (no `_duplicate` is done), and the application should no longer access the pointer directly. (If this is not the desired behavior, then the caller can explicitly `_duplicate` the object reference before assigning it to the sequence element.) However, when assigning to such an object reference sequence element from a `_var` object or from another struct, union, array, or sequence (rather than from an object reference pointer), a `_duplicate` is done automatically.

For a storage-managing sequence whose elements are Strings, when assigning a value to such an element or deleting the sequence, any previously held (non-null) `char*` is automatically freed. As when assigning to `String_var`s, assigning a `char*` to a string element does not make a copy, but assigning a `const char*`, a `String_var`, or another struct/union/array/sequence String member does automatically make a copy. Thus, one should never assign a string literal (such as "abc") to such an element without an explicit cast to `const char*`. When assigning a `char*` that occupies static storage (rather than one that was dynamically allocated), the caller can use `CORBA::string_dup` to duplicate the string before assigning it.

There is a corresponding `_var` type defined for every sequence class. The `_var` type for a sequence provides an overloaded operator[] that forwards the operator to the underlying sequence.

Following is an example that illustrates loading and accessing the elements of a sequence. This example illustrates a recursive sequence (whose entries are structs of the same type that contain the sequence). The IDL for the example is shown below:

```
struct S
{
    long sf1;
    sequence sf2;
};
typedef sequence Sseq;
```

The following is an example program that creates and loads a sequence of type `Sseq` and then prints out its contents.

```
#include seq_C.cpp
#include stdio.h
main()
{
    int i,j;
    Sseq seq;
    seq.length(3); // set length of seq to 3
    for (i=0; i<3; i++) { // index the three S structs in seq
        seq[i].sf1 = i; // place a number in the i-indexed struct
        seq[i].sf2.length(i+1); // set length of the sequence in
        // the i-indexed struct
        for (j=0; j<i+1; j++) // index the i+1 S structs in the sequence
            // in the i-indexed struct
            seq[i].sf2[j].sf1 = (i+1)*10+j; // place a number in
            // the j-indexed struct
    }
    // OK. Print out what you have created!
    printf("seq = (%d sequence elements)\n", seq.length());
    for (i=0; i<3; i++)
    {
        printf("    struct[%d] = {\n", i);
        printf("        sf1 = %d\n", seq[i].sf1);
        printf("        sf2 = (%d sequence elements)\n",
            seq[i].sf2.length());
        for (j=0; j<i+1; j++)
        {
```

```

        printf("        struct[%d] = \n",j);
        printf("            sf1 = %d\n", seq[i].sf2[j].sf1);
        printf("            sf2 = (%d sequence elements)\n",
            seq[i].sf2[j].sf2.length());
        printf("        }\n");
    }
    printf("    }\n");
}

```

Note that the above program never explicitly constructs any data of type S, even though the sequences contain structs of this type. The reason is that when a sequence buffer is allocated, default constructors are run for each of the buffer elements. So, when the above program sets the length of a sequence of S structs (either at the top level for the seq variable, or for the sf2 field of an S struct in seq), the resulting buffer is automatically populated with default structs of type S.

The output from the above program is:

```

seq = (3 sequence elements)
  struct[0] = {
    sf1 = 0
    sf2 = (1 sequence elements)
      struct[0] = {
        sf1 = 10
        sf2 = (0 sequence elements)
      }
  }
  struct[1] = {
    sf1 = 1
    sf2 = (2 sequence elements)
      struct[0] = {
        sf1 = 20
        sf2 = (0 sequence elements)
      }
      struct[1] = {
        sf1 = 21
        sf2 = (0 sequence elements)
      }
  }
  struct[2] = {
    sf1 = 2
    sf2 = (3 sequence elements)
      struct[0] = {
        sf1 = 30
        sf2 = (0 sequence elements)
      }
      struct[1] = {
        sf1 = 31
        sf2 = (0 sequence elements)
      }
      struct[2] = {
        sf1 = 32
        sf2 = (0 sequence elements)
      }
  }
}

```

C++ bindings for CORBA Strings

The mapping for strings is provided by corba.h, within the CORBA scope. See ["IDL name scoping" on page 2](#) for more information. The user-visible types are CORBA::String and CORBA::String_var. CORBA::String is a typedef name for char*. The CORBA::String_var class performs storage management of a dynamically allocated CORBA::String. The following functions are for dynamic allocation/deallocation of memory to hold a String:

- CORBA::string_alloc
- CORBA::string_free
- CORBA::string_dup

A String_var object behaves as a char*, except that when it is assigned to, or goes out of scope, the memory it points to is automatically freed by CORBA::string_free. When a String_var is constructed or assigned from a char*, the String_var assumes ownership of the string and the caller should no longer access the string directly. (If this is not the desired behavior, as when the char* occupies static storage, the caller can use CORBA::string_dup to copy the char* before assigning it.) When a String_var is constructed or assigned from a const char*, another String_var, or a String element of a struct, union, array, or sequence, an automatic copy of the source string is done. The String_var class provides subscripting operations to access the characters within the embedded string.

C++ compilers do not treat a string literal (such as "abc") as a const char* upon assignment;

given both a const and a non-const assignment operator, the compiler will choose the non-const operator. As a result, when assigning a string literal to a String_var, no copy of the string into dynamically allocated memory is made; the pointer "owned" by the String_var will point to memory that cannot be freed. Thus, string literals should not be assigned to a String_var without an explicit cast to const char*.

Some examples using String_var objects are:

```
// first some supporting functions for the examples
char* f1()
{
    return "abc";
}
char* f2()
{
    char* s=CORBA::string_alloc(4);strcpy(s,"abc");return s;
}
// then the examples
void main()
{
    CORBA::String_var s1;
    if (0) s1 = f1();// Wrong!! The pointer cannot be freed and
    // no copy is done.
    if (0) s1 = "abc"; // Also wrong, for the same reason.
    const char* const_string = "abcd"; // *const_string cannot be changed
    s1 = const_string; // OK. A copy of the string is made because
    // it is const, and the copy can be freed.
    CORBA::String_var s3 = f2();// OK. no copy is made, but f2
    // returns a string that can be freed
    CORBA::String_var s4 = CORBA::string_alloc(10); // also OK. no copy
    s4 = s3; // s4 will use string_free followed by string_dup
    long l4 = strlen(s4); // l4 will receive 3
    long l1 = strlen(s1); // l1 will receive 4
    if (l4 >= l1)
        strcpy(s4,s1); // OK, but only because of the condition.
    // note that s4's buffer only has size=4.
    s4 = const_string; // OK. s4 will use string_free followed by
    // string_dup. The copy is made because String_vars
    // must reference a buffer that can be modified.
}
// The s1, s3 and s4 destructors run successfully, freeing their buffers
```

C++ bindings for CORBA Struct types

An IDL struct type is mapped to a corresponding C++ struct whose field names correspond to those in the IDL declaration, and whose field types support access and storage of the C++ types corresponding to the IDL struct field types. Dynamically allocated storage used to hold such a C++ struct must be allocated and freed using the C++ new and delete operators.

When a new struct is created, the default constructor for each of its fields implements. Object reference fields are initialized to nil references, and String fields are initialized to NULL. When the struct is deleted (or goes out of scope), the destructor for each of its fields implements. The (default) copy constructor performs a deep copy, including duplicating object references; the (default) assignment operator acts as the destructor followed by the copy constructor.

The actual types of the fields in the C++ struct to which an IDL struct is mapped may be auxiliary classes for the purpose of storage management. In particular, String and object reference field types are auxiliary classes that manage Strings and object references in the same way that the associated _var classes do. Although client code should not depend on the names of these auxiliary classes, the client code does need to know that struct fields containing Strings and object references are managed by these auxiliary classes.

When assigning a value to a struct field that is an object reference, the assignment operator for the struct field will automatically release the previous value (if any). When assigning an object reference pointer to a struct member, the struct member assumes ownership of the pointer (no _duplicate is done), and the application should no longer access the pointer directly. (If this is not the desired behavior, then the caller can explicitly _duplicate the object reference before assigning it to the struct member.) However, when assigning to an object reference struct member from a _var object or from another struct, union, array, or sequence member (rather than from an object reference pointer), a _duplicate is done automatically.

When assigning a value to a struct field that is a String, or when the struct is deleted or goes out of scope, any previously held (non-null) String is automatically freed. As when assigning to String_vars, assigning a char* to a String field does not make a copy, but assigning a const char *, a String_var, or another struct/union/array/sequence String member does automatically make a copy. One should never assign a string literal (for example, "abc") to a String struct member without an explicit cast to "const char*". When assigning a char* that occupies static storage (rather than one that was dynamically allocated), the caller can use CORBA::string_dup to duplicate the string before assigning it.

As with all constructed types, a _var type is provided for managing an instance of the C++ struct that corresponds to an IDL struct. When assigning one struct's _var to another, the receiving _var deletes its current pointer (thus running all contained destructors), and creates a new struct to hold the assignment result, which is initialized using copy constructors for each of the contained fields. Thus, for example, if the source struct has an object reference field, the struct _var assignment will automatically duplicate this reference.

The IDL that follows is used in the succeeding example, which shows both correct and incorrect ways to create and manipulate the corresponding C++ struct and the corresponding _var type :

```
Interface A
{
    struct S
    {
        string f1;
        A      f2;
    };
};
```

The following code illustrates both correct and incorrect ways to create and manipulate the corresponding C++ struct and the corresponding _var type.

```
{
    A::S_var sv1 = new A::S;
    A::S_var sv2 = new A::S;
    // sv1->f1 = "abc"; -- Wrong! f1 cannot free this pointer later
    sv1->f1 = CORBA::string_alloc(20);
    A_ptr a1 = // get an A somehow
    A_ptr a2 = // get an A somehow
    sv1->f2 = a1; // a1 still has ref cnt = 1
    sv2->f1 = CORBA::string_alloc(20);
    sv2->f2 = a2; // a2 still has ref cnt = 1
    sv1 = sv2; // This runs copy ctors, and increments a2's ref cnt.
    // Also, a1's ref count is decremented.
    sv1->f1 = sv2->f1;
}
```

C++ bindings for CORBA Union types

Union fields are not directly accessible to C++ programmers. Instead, the C++ mapping for IDL unions defines a class that provides accessor methods for the union discriminator and the corresponding union fields. The union discriminator accessor is named `_d`. The union field accessors are named using the IDL union field names and are overloaded to allow both reading and writing.

Note: A deviation from the CORBA specifications exists; there is no support of longlong discriminators in unions.

Setting a union's value using a field accessor automatically sets the discriminator, and releases the storage associated with the previous value, if any. It is an error for an application to attempt to access the union's value through an accessor that does not match the current discriminator value. It is also an error for the application to use the discriminator modifier method to implicitly switch between difference union members.

Unions with implicit default members (those that do not have an explicit default case and do not list all possible values of the discriminator as cases) provide a `_default` method, for setting the discriminator to a legal default value. This method causes the union's value to be composed only of the legal default value, because there is no explicit default member in this case.

A `_var` type is defined, for managing a pointer to a union in dynamically allocated memory.

To illustrate the C++ bindings for IDL unions, consider the following IDL:

```
module A
{
    interface X
    {
    };
    union U switch (long)
    {
        case 1: long u1;
        case 2: string u2;
        case 3: X u3;
    };
};
```

The following code illustrates usage of the C++ bindings corresponding to the previous IDL:

```
{
    X_ptr x = // get an X somehow
    A::U_var uv = new A::U;
    uv.u2((const char*) "testing"); // sets the discriminator to 2
    // and copies the string
    if (u._d() == 2) // the condition evaluates to true
        u.u1(23); // frees the string, and sets the discriminator to 1
    if (u._d() == 1) // the condition evaluates to true
        u.u3(x); // duplicates x and sets discriminator to 3
}
```

The default constructor of a union class does not initialize the discriminator or the union members, so the application must initialize the union before accessing it. The copy constructor and assignment operator perform deep copies. The assignment operator and destructor release all storage owned by the union.

With respect to memory management, accessor and modifier methods for union members work similarly to those for struct members. Modifier methods make a deep copy of their input when passed by value (for simple types) or by reference (for constructed types). Accessor methods that return a non-const reference can be used by the application to update a union member's value, but only for struct, union, sequence, and any members.

The modifier method for a string union member makes a copy when given a `const char*` or a `String_var`, but not when given a `char*`. As shown in the example above, a string literal should not be assigned to a union without an explicit `"const char"` cast. The accessor method for a string union member returns a `const char*`, therefore the string union member cannot be modified. (This is done to prevent the string union member from being assigned to a `String_var`, resulting in memory management errors.)

The modifier method for an object reference union member always duplicates the input object reference and releases the previous object reference value, if any. The accessor method for an object reference union member does not duplicate the returned object reference, because the union retains ownership of it.

The accessor method for an array union member returns a pointer to the array slice. The application can thus read or write the union-member array elements using subscript operators. If the union member is an anonymous array (one without an explicit type name), the union defines (typedefs) the slice type, by concatenating a leading underscore and appending `"_slice"` to the union member name.

C++ bindings for CORBA WStrings

The `WString` type provides support for wide strings. It is fairly comparable to using strings except for type declarations and assignments:

```
#include wctr.h // For WChar and WString support
...
const wchar_t* wcomments = L"This policy looks pretty good...";
wchar_t* wcommentsResult=:CORBA:wstring_alloc(wcslen(wcomments));
=:CORBA::WString_var wcommentsResult_var(wcommentsResult);
policyVar->wcomments(wcomments);
if (!wcscmp(wcommentsResult_var, wcomments) )
```

```

{
    cout << "Wcomments not set" << endl;
    return 1;
}
else
{
    cout << "Wcomments set correctly..." << endl;
}
}
wcommentsResult = policyVar->wcomments();

```

CORBA C++ binding restrictions

When a forward reference to an interface appears within an IDL module, the IDL compiler issues an error message if the referenced interface is not defined within the module. When a similar unresolved forward reference appears at global (file) scope, a warning is issued that indicates the bindings being emitted will not include a mapping for the undefined interface. For information on the scope see ["IDL name scoping" on page 2](#). The assumption is that the interface will be defined by other bindings than those being currently generated. This approach supports IDL files with mutually-referential interfaces (as long as they appear at global scope). The following example illustrates how to organize the IDL files for such cases:

```

// file foo.idl
#ifndef foo_idl
#define foo_idl
interface Foo; // declare Foo so bar.idl can refer to it
#include bar.idl
interface Foo
{
    Bar fool(); // notice the use of Bar
};
#endif // foo_idl
// file bar.idl
#ifndef bar_idl
#define bar_idl
interface Bar; // declare Bar so foo.idl can refer to it
#include foo.idl
interface Bar
{
    Foo bar1(); // notice the use of Foo
};
#endif // bar_idl

```

Due to problems inherent to the CORBA 2.1 mapping for C++, there are currently two known limitations with respect to handling legal CORBA 2.1 IDL. The compiler provides informative error messages in these two cases, and indicates that C++ bindings cannot be generated. The cases are:

- The C++ bindings map most IDL data types to C++ classes contained within a nesting scope provided by another C++ class. However, it is not legal to define a nested C++ class (or any other type) that has the same name as a containing C++ class. Thus, for example, the following IDL cannot be mapped to useful C++ bindings:

```

module X
{
    interface X ...;
    // or struct X ...;
    // or union X ...;
    // or typedef sequence < > X;
    // ...
};

```

- The C++ bindings map attributes into overloaded C++ accessor functions whose name is the attribute name. As a result, for example, the following IDL will not map to useful C++ bindings (because Y's l method interferes with the inherited mapping for X's attribute). If Y's method took any arguments, there would not be a problem, because of C++ overloading. The compiler indicates an error only when C++ overloading will not distinguish inherited accessors from newly introduced methods (or vice versa).

```

interface X
{
    attribute long l;
}
interface Y : X
{
    long l();
};

```

CORBA programming: Name scoping and modules in the C++ bindings

IDL scoped names are mapped to C++ scopes as follows.

- In the IBM C++ bindings, IDL modules are, by default, mapped to C++ classes of the

same name. If the programmer using the bindings #defines `_USE_NAMESPACE` before including the bindings, then the bindings map the IDL module to a C++ namespace of the same name. IDL definitions occurring within a module are mapped to corresponding C++ definitions within the C++ module class or namespace.

- IDL interfaces are mapped to C++ classes. All IDL constructs defined within an interface are mapped to corresponding C++ definitions within the C++ interface class.
- Every use in IDL of a C++ keyword (such as "class") is mapped into the same identifier with a leading underscore.

Commonly used CORBA interfaces

The most commonly used CORBA interfaces are described in the following topics:

- ["CORBA class interfaces" on page 40](#)
- ["CORBA::object interfaces" on page 40](#)
- ["CORBA::ORB interfaces" on page 41](#)

For more information on operations defined by these interfaces, refer to the ORB section within the CORBA module of the *Programming Reference*.

CORBA class interfaces

The CORBA interface provides the following commonly used class operations. These are used like a C++ class reference (for example `CORBA::is_nil(somePointer);`).

is_nil

This operation returns a boolean that indicates if the input object reference is nil.

This is useful for many operations involving object references, including those operations that do not throw exceptions when they fail - for example

```
CORBA::Object::_narrow().
```

release

This operation releases resources associated with an object or pseudo-object reference. This operation may or may not perform a C++ delete operation. A reference count is used by this operation and `CORBA::Object::_duplicate()`. When the reference count reaches zero then the appropriate delete operations are performed. Care must be taken when using the `release` and `_duplicate` operations to ensure that objects are not leaked or inadvertently deleted. Alternatively use the `_var` technique described for **string_dup** below.

string_dup

This operation copies a string. A common example of its use is when returning a string from an operation. Strings and wide strings, unlike the other basic CORBA types, have associated allocated memory. So care must be taken when using these variables. The resulting string should subsequently be freed by using the `CORBA::string_free` operation, or by assigning the string to a `_var` variable which will free the string appropriately.

CORBA::object interfaces

The CORBA interface provides the following object interfaces:

_duplicate

This operation duplicates an object reference. This is particularly useful when passing references to objects to resolve memory ownership issues. For every `_duplicate` that is performed on an object an equal number of `release()` must also be performed for proper memory management. An alternative to the `_duplicate()` and `release()` logic is to use `_var` support as described for **string_dup** in ["CORBA class interfaces" on page 40](#).

`_is_a`

This operation is used to determine whether an object reference supports a given IDL interface. If the object supports the interface the `_narrow` operation can be successfully performed.

`_is_equivalent`

This operation is used to determine whether two object references refer to the same object.

`_narrow`

This operation is used to narrow a more generic interface to a more specific interface. This operation will return an empty pointer without throwing an exception if the interface cannot be narrowed to the requested type. Care must be taken to check the returned value before using it.

`_nil`

This operation returns a nil `CORBA::Object`. This object could be used for comparison operations.

`_non_existent`

This operation determines whether an object reference refers to a valid object. This will result in verification of the object reference only, no other operations are performed on the requested object.

CORBA::ORB interfaces

The CORBA interface provides the following ORB interfaces:

`object_to_string`

This operation converts an object reference to an external form that can be stored for later use or exchanged between processes. The `string_to_object` operation can be used to reconstruct the object reference.

`string_to_object`

This operation converts a stringified object reference to a reconstructed object reference. The `object_to_string` operation must have been used to create the input stringified data.

Note: Although `object_to_string` is the way to save object references for future usage, the returned data should only be used with `string_to_object` to reconstruct that object reference. Do not use the string for comparing equivalence of object references. The `object_to_string` operation may return different values at different times because various Object Services may be adding information to this IOR.

C++ bindings for CORBA interfaces

The CORBA 2.1 C++ client bindings define a variety of C++ types corresponding to a single IDL interface. For example, an IDL interface `I` is mapped to C++ types with the following four names:

- `I`
- `I_ptr`
- `IRef`
- `I_var`

The types named `I` and `I_var` are classes. The types `I_ptr` and `IRef` are unspecified by CORBA, but are required to name the same type; these types are the C++ form for an object reference.

Note: The use of the `IRef` types will be removed by the CORBA 2.1 specification.

The class `I` is referred to as the interface class corresponding to IDL interface named `I`; the C++ mappings of the typedefs, operations, and constants defined within the IDL interface `I` appear publicly within the C++ interface class `I`. For example, an IDL operation that accepts an in parameter of interface type `I` is mapped to a C++ virtual member function of the class named `I` that has a parameter of type `I_ptr`. Similarly, an IDL operation in `I` that returns data of type `I` is mapped to a C++ member function in the class `I` that returns data of type `I_ptr`.

As with other user-defined IDL types, the `I_var` type is used to assist storage management. Specifically, an `I_var` type holds an `I_ptr` and can be used as if it were an `I_ptr`. When an `I_var` type is assigned a new value or when it goes out of scope, it releases the `I_ptr` it is holding at that time.

The CORBA 2.1 specification prohibits CORBA-compliant applications from:

- Explicitly creating an instance of an interface class, as in:

```
I my_instance; // NOT ALLOWED!  
I_ptr my_instance = new I; // NOT ALLOWED!
```

- Declaring a pointer (`I*`) or a reference (`I`) to an interface class.

Instead, the `I_ptr`, `IRef`, and `I_var` types must be used to hold object references, and object references can only be created (by client applications) by invoking methods that return object references. The interface class `I` is used by client applications only as a name scope.

IDL operations defined in (or inherited by) interface `I` are invoked in C++ using the arrow (`->`) operator on either an `I_ptr`, `IRef`, or `I_var` type.

Nil object references of type `I_ptr` can be obtained using a static member function of `I` called `_nil()`. Operations cannot be invoked on nil object references. The `CORBA::is_nil` function is the only CORBA-conformant way to determine whether a given object reference is nil. `CORBA::release` can be invoked on a nil object reference, but is not needed. The `_duplicate` and `_narrow` functions defined by the C++ bindings can be given a nil object reference.

In the IBM C++ bindings, the CORBA-prescribed types are implemented as follows:

1. The interface class for `I` is derived using virtual inheritance from the interface classes for `I`'s IDL parents. When `I` has no IDL parents, its interface class is derived using virtual inheritance from `CORBA::Object`. Types, constants, and operations declared within the `I` interface are mapped to types, constants, and member functions declared within the corresponding interface class.
2. The object reference types `I_ptr` and `IRef` are typedef'd to `I*` (for example, an `I_ptr` points to an object of type `I`). However, CORBA 2.1 specifies that treating an `I_ptr` as a C++ pointer (e.g., using conversion to `void*`, arithmetic and relational operators, test for equality) is not conformant, although this is not enforced by the bindings.
3. An instance of `I` addressed is called a proxy, and is created by a proxy factory object of class `ProxyFactory`. For each interface `I`, the bindings define a `ProxyFactory` class, and provide a global instance of this class with the name `_ProxyFactory`.
4. Nil object references are represented as NULL pointers (but CORBA 2.1 conformant applications should not assume so, and should instead use the `_nil()` and `is_nil()` functions to manipulate nil object references).
5. The `I_var` class introduces an instance variable of type `I_ptr`. The purpose of an `I_var` object is to handle release operations on the `I_ptr` that it holds.
6. An auxiliary class `I_SeqElem` is used to return sequence elements, and is similar to the

`I_var` class. It is returned from array access operations on an IDL type sequence. An `I_SeqElem` is different from an `I_var` in that it must honor the release setting of the sequence from which it is selected (that is, it only owns the object that its `I_ptr` references if it was taken from a sequence that owns its buffer storage).

For more details on C++ bindings for CORBA interfaces, see the following topics:

- [“Managing object references” on page 43](#)
- [“Widening object references” on page 43](#)
- [“Narrowing object references” on page 43](#)
- [“Narrowing to a C++ implementation” on page 44](#)

Managing CORBA object references

The mapping for interface `I` defines a static member function named `_duplicate` that takes as input an object reference of type `I_ptr` and returns an object reference of type `I_ptr` (potentially the same reference, when reference counting is employed, as is the case with WebSphere Application Server C++ bindings). The `CORBA::release` function indicates that the caller will no longer access the object reference, and the resources associated with the object reference can be deallocated. (In the WebSphere Application Server C++ bindings, an object reference is only deleted when its reference count falls to zero, that occurs only if `CORBA::release` is called for each `_duplicate` or `_narrow` performed on the object reference.)

Duplicating an object reference using `_duplicate` is analogous to copying a string before transferring ownership of it, and releasing an object reference is analogous to deleting a string that is no longer needed. Unlike strings, object references cannot be directly copied or deleted by the client programmer; object references are managed by the ORB and can only be duplicated or released by the application.

Widening CORBA object references

If interface `A` is a (direct or indirect) base of interface `B`, the following assignments do not require an explicit C++ cast:

- `B_ptr` to `A_var`
- `B_ptr` to `A_ptr`
- `B_ptr` to `Object_var`
- `B_ptr` to `Object_ptr`
- `B_var` to `A_ptr`
- `B_var` to `Object_ptr`

`B_var` cannot be assigned to `A_var` or a compile-time error occurs. To assign `B_var` to `A_var`:

- Use `B::_duplicate` on `B_var` to create `B_ptr`.
- Assign `B_ptr` to `A_var`.

Narrowing CORBA object references

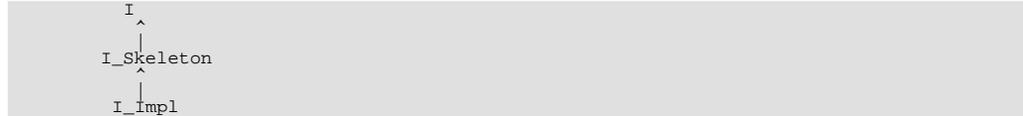
The mapping for an interface `I` defines a static member function named `_narrow` that takes as input an object reference of any type (for example, an `Object_ptr`) and returns an object reference of type `I_ptr`. If the referenced object (the actual implementation object corresponding to the proxy addressed by the input object reference) does not support the `I` interface, the result is `NULL`; otherwise, the `I_ptr` addresses an object that also supports the `I` interface. In the case where the proxy addressed by the input argument does not support interface `I` and the actual implementation object does, the `I_ptr` returned by `I::_narrow` addresses a different proxy object than the input argument.

The `_narrow` static member function does an implicit `_duplicate` of the input argument.

Therefore, the caller is responsible for releasing both the object reference input to `_narrow` and the return result.

CORBA programming: narrowing to a C++ implementation

Given an interface pointer to an object, it is sometimes useful to narrow to the implementation pointer of the object. For example, given interface `I`, the C++ implementation hierarchy for `I` might look like:



You might want to convert a pointer to `I` into a pointer to `I_Impl`. There is no CORBA-prescribed mechanism for this conversion. Within the confines of the C++ language, dynamic cast can be used.

CORBA programming: Storage management and `_var` types

The C++ bindings try to make the programmer's storage management responsibility as easy as possible. One aspect of this is the "`_var`" types. For each user-defined structured IDL type `T` (struct, union, sequences, and arrays) and for interfaces, the bindings generate both a class `T` and a class `T_var`. The classes `CORBA::String` and `CORBA::Any` also have corresponding `CORBA::String_var` and `CORBA::Any_var` classes.

The essential purpose of a `_var` object is to hold a pointer to dynamically allocated memory. A `_var` object can be used as if it were a pointer to the IDL type for which it is named; special constructors, assignment operators, and conversion operators make this work in a way that is invisible to programmers. The memory pointed to by a `_var` object is always considered to be owned (managed) by the `_var` object, and when the `_var` object is deleted, goes out of scope, or is assigned a new value, it deletes (or, in the case of an object reference, releases) the managed memory.

A typical `_var` object is declared by a programmer as an automatic (stack) variable within a code block, and is then used to receive an operation result or is passed to an operation as an out parameter. Later, when the code block is exited, the `_var` object destructor runs and its managed memory is deleted (or, for object references, released).

When a pointer (rather than another `_var` object or struct/union/array/sequence element) is assigned to (or used to construct) a `_var` object, this pointer should point to dynamically allocated memory, because the `_var` object does not make a copy; it assumes ownership of the pointer and will later delete it (or, for object references, release it). The single exception is that pointers to const data can be assigned to a `_var` object. When this occurs, the `_var` object dynamically allocates new memory and copies the const data into the new memory. A pointer assigned to a `_var` object must not be "owned" by some other data structure, and the pointer should not be subsequently used by the application except by the `_var` object.

The default constructor for a `_var` type loads the contained pointer with `NULL`. You must assign a value to a `_var` object created by a default constructor before invoking methods on it, just as you must assign a value to a pointer variable before invoking methods on it.

The copy constructor and `_var` assignment operator of a `_var` type perform a deep copy of the source data. The copy is later deallocated (or released, in the case of object references) when the `_var` is destroyed or when it is assigned a new value.

The following is the typical form for a `T_var` class, emitted for an IDL--constructed data type named `T`:

```

class T_var
{
public:
    T_var ();
    T_var (T*);
    T_var (const T_var&);
    ~T_var ();
    T_var operator= (T*);
    T_var operator= (const T_var&);
    T * operator-> const ();
    ...
};

```

For more information on storage management and argument passing see [“Argument passing considerations for C++ bindings”](#) on page 45.

CORBA programming: Argument passing considerations for C++ bindings

Rules must be observed when passing parameters to a CORBA object implementation. The type used to pass the parameters of a method signature is dependent on the IDL type and the directionality of the parameter (in , inout, out, or return value).

The following rules for passing these parameters are dictated by CORBA OMG IDL to C++ mapping, and must be followed to:

- Ensure the required access authority.
- Prevent memory leaks.
- Ensure that the allocation and deallocation of memory is performed consistently.

in parameters

The caller (client) must allocate the input parameters. The callee (implementation) is restricted to read access. The caller is responsible for the eventual release of the storage. Primitive types and fixed-length aggregate types may either be heap allocated or stack allocated. By their nature, variable-length aggregates cannot be completely stack allocated.

inout parameters

For inout parameters, the caller provides the initial value and the callee may change that value. For primitive types and fixed-length aggregates this is straight forward. The caller provides the storage and the callee overwrites the storage on return. For variable-length aggregates the size of the contained data provided on input may differ than the size of the contained data provided on output. Therefore, the callee is required to deallocate any input contained data that is being replaced on output with callee allocated data. For object references, the caller provides an initial value: if the callee reassigns the value the callee must first release the original input value. The callee assumes or retains ownership of the returned parameters and must eventually deallocate or release them.

out parameters

For primitive types and fixed-length aggregate types, the caller allocates the storage for the out parameter and the callee sets the value. For variable-length aggregate types, the caller allocates a pointer and passes it by reference and the callee sets the pointer to point to a valid instance of the parameter's type. For object references the caller allocates storage for the `_ptr` and the callee sets the `_ptr` to point to a valid instance. Because a pointer to an array in C++ must actually be represented as a pointer to the array element type, CORBA defines an `array_slice` type, where a slice is an array with all the dimensions of the original except the first. The output parameter is typed as a reference to an `array_slice` pointer. The caller allocates the storage for the pointer and the callee updates the pointer to point to a valid instance of an `array_slice`. The caller assumes or retains ownership of the output parameter storage and must eventually deallocate it or, in the case of object references, release it.

return values

For primitive types and fixed-length aggregate types, the caller allocates the storage for the return value and the callee returns a value for the type. For variable-length aggregate types, the caller allocates a pointer and the callee returns a pointer to an instance of the type. For object references the caller allocates storage for the `_ptr` and the callee returns a `_ptr` that points to a valid object instance. As a pointer to an array in C++ must actually be represented as a pointer to the array element type, the `array_slice` type is used for returning array values. The caller allocates storage for a pointer to the `array_slice` and the callee returns a pointer to a valid instance of an `array_slice`. The caller assumes or retains ownership of the storage associated with returned values and must eventually deallocate it or, in the case of object references, release it.

These rules for passing parameters are captured in and enforced by the header files produced when an IDL interface description is compiled. Some rules cannot be enforced by the bindings. For example, parameters that are passed or returned as a pointer type (`T*`) or reference to pointer (`T*&`) should not be passed or returned as a null pointer. Memory management responsibilities cannot be enforced by the bindings. Client (caller) and implementation (callee) programmers must understand and implement according to these rules.

For more detailed information on storage management and argument passing see the following topics:

- [“C++ type mapping for argument passing” on page 46.](#)
- [“Storage management responsibilities for arguments” on page 48.](#)

CORBA programming: C++ type mapping for argument passing

Argument type mappings are discussed in this topic and summarized in the two tables below. For the rules that must be observed when passing parameters (in , inout, out, or return value) to a CORBA object implementation, see [“Argument passing considerations for C++ bindings” on page 45](#) .

For primitive types and enumerations, the type mapping is straightforward. For in parameters and return values the type mapping is the C++ type representation (abbreviated as "T" in the text that follows) of the IDL specified type. For inout and out parameters the type mapping is a reference to the C++ type representation (abbreviated as "T&" in the text that follows).

For object references, the type mapping uses `_ptr` for in parameters and return values and `_ptr&` for inout and out parameters. That is, for a declared interface A, an object reference parameter is passed as type `A_ptr` or `A_ptr&`. The conversion functions on the `_var` type permit the client (caller) the option of using `_var` type rather than the `_ptr` for object reference parameters. Using the `_var` type may have an advantage in that it relieves the client (caller) of the responsibility of deallocating a returned object reference (out parm or return value) between successive calls. This is because the assignment operator of a `_ptr` to a `_var` automatically releases the embedded reference.

The type mapping of parameters for aggregate types (also referred to as complex types) are complicated by when and how the parameter memory is allocated and deallocated. Mapping in parameters is straightforward because the parameter storage is caller allocated and read only. For an aggregate IDL type t with a C++ type representation of T the in parameter mapping is `const T&`. The mapping of out and inout parameters is slightly more complex. To preserve the client capability to stack allocate fixed length types, OMG has defined the mappings for fixed-length and variable-length aggregates differently. The inout and out mapping of an aggregate type represented in C++ as T is `T&` for fixed-length aggregates and as `T*&` for variable-length aggregates.

Table: 1. Basic argument and result passing

| Data Type | In | Inout | Out | Return |
|----------------------|-----------------|-------------|---------------|--------------|
| short | short | short& | short& | short |
| long | long | long& | long& | long |
| unsigned short | ushort | ushort& | ushort& | ushort |
| unsigned long | ulong | ulong& | ulong& | ulong |
| float | float | float& | float& | float |
| double | double | double& | double& | double |
| boolean | boolean | boolean& | boolean& | boolean |
| char | char | char& | char& | char |
| wchar | wchar | wchar& | wchar& | wchar |
| octet | Octet | Octet& | Octet& | Octet |
| enum | enum | enum& | enum& | enum |
| object reference ptr | objref_ptr | objref_ptr& | objref_ptr& | objref_ptr |
| struct, fixed | const struct& | struct& | struct& | struct |
| struct, variable | const struct& | struct& | struct*& | struct* |
| union, fixed | const union& | union& | union& | union |
| union, variable | const union& | union*& | union*& | union* |
| string | const char* | char*& | char*& | char* |
| wstring | const char* | char*& | char*& | char* |
| sequence | const sequence& | sequence& | sequence*& | sequence* |
| array, fixed | const array | array | array | array slice* |
| array, variable | const array | array | array slice*& | array slice* |
| any | const any& | any& | any*& | any* |

For an aggregate type represented by the C++ type T, the T_var type is also defined. The conversion operations on each T_var type allows the client (caller) to use the T_var type directly for any directionality, instead of using the required form of the T type (T, T& or T*&). The emitted bindings define the operation signatures in terms of the parameter passing modes shown in the table "[T_var argument and result passing](#)" on page 47, and the T_var types provide the necessary conversion operators to allow them to be passed directly.

Table: 2. T_var argument and result passing

| Data Type | In | Inout | Out | Return |
|----------------------|-------------------|-------------|-------------|------------|
| object reference_var | const object_var& | objref_var& | objref_var& | objref_var |
| struct_var | const struct_var& | struct_var& | struct_var& | struct_var |
| union_var | const union_var& | union_var& | union_var& | union_var |
| string_var | const | string_var& | string_var& | string_var |

| | | | | |
|--------------|------------------------|---------------|---------------|--------------|
| | string_var& | | | |
| sequence_var | const sequence_var& | sequence_var& | sequence_var& | sequence_var |
| array_var | const array_var& | array_var& | array_var& | array_var |

For parameters that are passed or returned as a pointer type (T*) or reference to pointer(T*&) the programmer should not pass or return a null pointer. This cannot be enforced by the bindings.

CORBA programming: Storage management responsibilities for arguments

The storage access and allocation responsibilities for argument passing are summarized in the two tables below. For the detailed rules that must be observed when passing parameters (in , inout, out, or return value) to a CORBA object implementation, see [“Argument passing considerations for C++ bindings” on page 45](#) .

As an overall requirement when allocating and deallocating argument storage, the storage allocation rules for the specific type must be followed. Specifically, for strings, sequences, and arrays or for aggregate types composed of these types, the associated memory allocation and deallocation functions must be used. For string types this means the use of `string_alloc()`, `string_dup()`, and `string_free()`, for sequence types this means the use of `allocbuf()` and `freebuf()` and for arrays this means the use of `T_alloc()`, `T_dup()` and `T_free()`. The memory deallocation responsibilities of the client can be minimized by stack allocation and the use of the `_var` types when that is possible. When an argument is passed or returned as a pointer type, a NULL pointer value should never be passed or returned.

Table: 1. Argument storage responsibilities

| Data Type | Inout | Out | Return |
|--------------------------|-------|-----|--------|
| short | 1 | 1 | 1 |
| long | 1 | 1 | 1 |
| unsigned short | 1 | 1 | 1 |
| unsigned long | 1 | 1 | 1 |
| float | 1 | 1 | 1 |
| double | 1 | 1 | 1 |
| boolean | 1 | 1 | 1 |
| char | 1 | 1 | 1 |
| octet | 1 | 1 | 1 |
| enum | 1 | 1 | 1 |
| object reference pointer | 2 | 2 | 2 |
| struct, fixed | 1 | 1 | 1 |
| struct, variable | 1 | 3 | 3 |
| union, fixed | 1 | 1 | 1 |
| union, variable | 1 | 3 | 3 |
| string | 4 | 3 | 3 |
| sequence | 5 | 3 | 3 |

| | | | |
|-----------------|---|---|---|
| array, fixed | 1 | 1 | 6 |
| array, variable | 1 | 6 | 6 |
| any | 5 | 3 | 3 |

For definitions of the numerical values in the above table, refer to the table below:

Table: 2. Argument passing cases

| Case | Description |
|------|--|
| 1 | Caller allocates all necessary storage, except that which may be encapsulated and managed within the parameter itself. For inout parameters, the caller allocates the storage but need not initialize it, and the callee sets the value. Function returns are by value. |
| 2 | 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. |
| 3 | 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: in order to do so, the caller must first copy the returned instance into a new instance, then modify the new instance. |
| 4 | For inout strings, the caller provides storage for both the input string and the char* pointing to it. Because the callee may deallocate the input strings 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. |
| 5 | For inout sequences and any's , 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 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 completion of a request, the caller is not allowed to modify any values in the returned storage: in order to do so, the caller must first copy the returned array instance into a new array instance, then modify the new instance. |

CORBA internationalization considerations

When you code CORBA applications for international use, consider the following issues:

- [“Initializing client programs” on page 51.](#)
- [“Character set restriction” on page 51.](#)
- [“Passing object references over multiple platforms” on page 51.](#)
- [“Using the OMG char data type in IDL files” on page 51.](#)

CORBA internationalization: Initializing client programs

All C++ clients should have their locale information set correctly. To do this, add the following as the first line executed in the program:

```
setlocale(LC_ALL, "");
```

CORBA internationalization: Character set restriction

When developing CORBA applications, use only the Portable Character Set (PCS) in your IDL string type parameters.

The PCS consists of the following characters:

```
0 1 2 3 4 5 6 7 8 9
: ; < = > ? @ [ \ ] ^ _ ` ' ~ { | } ! " # $ % & ( ) * + , - . / <space>
a b c d e f g h i j k l m n o p q r s t u v w x y z
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
```

CORBA internationalization: Passing object references over multiple platforms

When passing an object reference that is stored in a file from one platform to another, any stringified values must be passed with the appropriate code page conversion.

For example, when you transfer files between Windows NT and OS/390, you must use an ASCII-aware mechanism (such as FTP in ASCII mode, or an ASCII NFS mount). Do not use FTP in binary mode.

CORBA internationalization: Using the OMG char data type in IDL files

Successful data transportation between client and server can involve code set conversions. But CORBA limits the size of a `char` data item to one octet during transportation. So if any `char` data item is expanded to more than one octet in length during code set conversion, a `CORBA::DATA_CONVERSION` exception is generated.

Use the `char` data type for a parameter or return result only when the parameter or return result can contain data only from the [“Portable Character Set” on page 51](#) (PCS). Otherwise, use a `string` data type.

The CORBA module

The CORBA module, defined in orb.idl, encompasses the interfaces that make up the following programming elements:

- The CORBA-compliant ORB.
- The TypeCode library.
- The Interface Repository Framework (IR).

The interfaces within this module are intended to be used to write CORBA-compliant, distributed client-server applications, in which objects can be accessed across address spaces, even across different machines. These interfaces constitute a CORBA-compliant Object Request Broker (ORB), a standardized transport for distributed object interaction.

The TypeCode and Interface Repository (IR) interfaces contained in the CORBA module are intended to be used to write client applications using the Dynamic Invocation Interface (wherein the interfaces to be used by the client are not known at compile time). The TypeCode library provides run-time access to descriptions of IDL data types. The Interface Repository (IR) Framework allows run-time access to information specified in IDL.

The files relating to the CORBA module are listed in the table below.

Table: 1. Files for the CORBA module

| *** | AIX | Solaris | Windows NT Visual C++ |
|------------------------|------------------------|-------------------------|------------------------|
| module file name | orb.idl | | |
| Java package file name | not applicable | | |
| C++ Header file name | corba.h | | |
| Linker files | libsomoror.a (for ORB) | libsomoror.so (for ORB) | somororm.lib (for ORB) |
| libsomorir.a (for IR) | libsomorir.so (for IR) | somorirm.lib (for IR) | |

The portions of the CORBA module that can be referenced in application-specific IDL is contained in orb.idl. The C++ language mapping for the CORBA module is contained in corba.h. This file includes not only C++ mappings for the interfaces defined in orb.idl, but also C++ mappings for CORBA *pseudo-objects* (objects that cannot be accessed remotely nor referenced in application IDL, but which provide services used in-process by client and server applications).

For information on the syntax and definition of types within the CORBA module, see [“CORBA module: Types” on page 52](#).

For information on each of the many classes and interfaces within the CORBA module, see the related topics.

CORBA module: Types

The following code fragment shows the syntax and definition of the available types within the CORBA module:

```
typedef sequence<octet, 1024> ReferenceData;
```

```

typedef string ScopedName;
typedef string RepositoryId;
typedef string Identifier;
typedef string VersionSpec;
typedef sequence<InterfaceDef> InterfaceDefSeq;
typedef sequence<Contained> ContainedSeq;
typedef sequence<StructMember> StructMemberSeq;
typedef sequence<UnionMember> UnionMemberSeq;
typedef sequence<Identifier> EnumMemberSeq;
typedef sequence<ParameterDescription> ParDescriptionSeq;
typedef Identifier ContextIdentifier;
typedef sequence<ContextIdentifier> ContextIdSeq;
typedef sequence<ExceptionDef> ExceptionDefSeq;
typedef sequence<ExceptionDescription> ExcDescriptionSeq;
typedef sequence<RepositoryId> RepositoryIdSeq;
typedef sequence<OperationDescription> OpDescriptionSeq;
typedef sequence<AttributeDescription> AttrDescriptionSeq;
struct StructMember
{
    Identifier name;
    TypeCode type;
    IDLType type_def;
};
struct UnionMember
{
    Identifier name;
    any label;
    TypeCode type;
    IDLType type_def;
};
struct ModuleDescription
{
    Identifier name;
    RepositoryId id;
    RepositoryId defined_in;
    VersionSpec version;
};
struct ConstantDescription
{
    Identifier name;
    RepositoryId id;
    RepositoryId defined_in;
    VersionSpec version;
    TypeCode type;
    any value;
};
struct TypeDescription
{
    Identifier name;
    RepositoryId id;
    RepositoryId defined_in;
    VersionSpec version;
    TypeCode type;
};
struct ExceptionDescription
{
    Identifier name;
    RepositoryId id;
    RepositoryId defined_in;
    VersionSpec version;
    TypeCode type;
};
struct AttributeDescription
{
    Identifier name;
    RepositoryId id;
    RepositoryId defined_in;
    VersionSpec version;
    TypeCode type;
    AttributeMode mode;
};
struct ParameterDescription
{
    Identifier name;
    TypeCode type;
    IDLType type_def;
    ParameterMode mode;
};
struct OperationDescription
{
    Identifier name;
    RepositoryId id;
    RepositoryId defined_in;
    VersionSpec version;
    TypeCode result;
    OperationMode mode;
    ContextIdSeq contexts;
    ParDescriptionSeq parameters;
    ExcDescriptionSeq exceptions;
};
struct InterfaceDescription
{
    Identifier name;
    RepositoryId id;
    RepositoryId defined_in;
    VersionSpec version;
    RepositoryIdSeq base_interfaces;
};
enum TCKind
{
    tk_null, tk_void, tk_ushort, tk_ulong,
    tk_short, tk_long, tk_double, tk_char,
    tk_float, tk_double, tk_TypeCode, tk_Principal, tk_objref,
    tk_octet, tk_any, tk_TypeCode, tk_Principal, tk_objref,
    tk_struct, tk_union, tk_enum, tk_string,
    tk_sequence, tk_array, tk_alias, tk_except,
    tk_longlong, tk_ulonglong,
    tk_wchar, tk_wstring, tk_fixed
};

```

```

};
enum DefinitionKind
{
    dk_none,          dk_all,
    dk_Attribute,    dk_Constant, dk_Exception, dk_Interface,
    dk_Module,      dk_Operation, dk_Typedef,
    dk_Alias,       dk_Struct,    dk_Union,    dk_Enum,
    dk_Primitive,  dk_String,    dk_Sequence, dk_Array,
    dk_Repository  dk_Wstring
};
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
};
enum AttributeMode {ATTR_NORMAL, ATTR_READONLY};
enum OperationMode {OP_NORMAL, OP_ONeway};
enum ParameterMode {PARAM_IN, PARAM_OUT, PARAM_INOUT};

```

CORBA module: AliasDef Interface

| | |
|---|--|
| Overview | Used by the Interface Repository to represent an OMG IDL definition that aliases another definition. |
| File name | somir.idl |
| Local-only | True |
| Ancestor interfaces | "TypedefDef Interface" on page 228 |
| Exceptions | "CORBA::SystemException" on page |
| Supported operations | "AliasDef::original_type_def" on page 54 |
| "IDLType::type" on page 127 | |

Intended Usage

An instance of an AliasDef object is used within the Interface Repository to represent an OMG IDL type that aliases another type. The AliasDef is typically used within the Interface Repository to represent a typedef statement as defined in OMG IDL. An instance of an AliasDef object can be created via the create_alias operation of the Container interface.

IDL syntax

```

module CORBA
{
    interface AliasDef:TypedefDef
    {
        attribute IDLType original_type_def;
    };
};

```

AliasDef::original_type_def

Overview

The original_type_def read and write operations provide for access and update of the type being aliased by an OMG IDL alias definition (AliasDef) in the Interface Repository.

| | |
|---------------------------|--|
| Original interface | "AliasDef Interface" on page 54 |
| Exceptions | "CORBA::SystemException" on page |

The original_type_def attribute identifies the type being aliased. Read and write operations are provided with parameter definitions as defined below.

IDL Syntax

```
attribute IDLType original_type_def;
```

Read operations

Input parameters

none

Return values

CORBA::IDLType_ptr

The returned pointer references an IDLType that represents the type aliased by the AliasDef. The memory is owned by the caller and can be released by invoking CORBA::release.

Write operations

Input parameters

CORBA::IDLType_ptr original_type_def

This parameter is used to modify the type aliased within the alias definition. Setting the original_type_def attribute also updates the inherited type attribute.

Return values

none

Example

```
// C++
// assume that 'this_alias' and 'this_struct'
// have already been initialized
CORBA::AliasDef * this_alias;
CORBA::StructDef * this_struct;
// change 'this_alias' to be an alias for 'this_struct'
this_alias-> original_type_def (this_struct);
// obtain the aliased type from the alias definition
CORBA::IDLType * returned_aliased_type;
returned_aliased_type = this_alias-> original_type_def ();
```

CORBA module: Any Class

| | |
|--------------------------|--|
| Overview | Represents a value having an arbitrary data type. |
| File name | any.h |
| Supported methods | "Any::_nil" on page 57 |
| | "Any::operator<<" on page 57 |
| | "Any::operator>>" on page 58 |
| | "Any::replace" on page 60 |
| | "Any::type" on page 60 |

Intended Usage

The Any class constitutes the C++ mapping of the IDL data type "any". An Any object can be used by a client or server application to represent application data whose type is not known at compile time. The Any contains both a data structure and a TypeCode that describes the data structure.

The Any class provides a non-default constructor whose parameters are: a CORBA::TypeCode_ptr (describing the type of data held by the Any), a void* pointer (the value to be contained by the Any) and a CORBA::Boolean indicating whether the Any is to assume ownership of the data. (The Any always duplicates the TypeCode rather than assuming ownership of the original.) After the Any assumes ownership of a value, the application should make no assumptions about the continued lifetime of the value. The default value for the Boolean flag is zero (indicating that the Any does not assume ownership of the value). The void* pointer given to the Any non-default constructor can be NULL.

The default constructor creates an Any with a tk_null TypeCode and no value. The copy

constructor and assignment operator for Any perform deep copies of both the TypeCode and the value contained by the Any being copied.

The Any class provides insertion (<<=) and extraction (>>=) operators that allow it to hold data of simple IDL types, while maintaining type safety (preventing one from creating an Any whose TypeCode and value do not match). The operators are also convenient because they alleviate the programmer from manipulating TypeCodes; the programmer simply streams data structures into or out of the Any, and the TypeCode is implied by the C++ type of the value being inserted or extracted.

Since the IDL boolean, octet, and char types do not map to distinct C++ types, the Any class introduces helper types for each, to allow each IDL type to have distinct insertion and extraction operators. These types (from_boolean, to_boolean, from_octet, to_octet, from_char, and to_char) are shown in the Types section below. To insert a boolean, octet, or char into an Any, or to extract one from an Any, construct a helper object (by passing the data to be inserted/extracted to the helper's constructor), then use the helper object with the corresponding Any insertion/extraction operator.

Similarly, because both bounded and unbounded strings in IDL map to char* in C++, the to_string and from_string helper types are introduced (see below) for inserting/extracting both bounded and unbounded strings to/from an Any. (Unbounded strings are signified by constructing the to_string or from_string helper with a bound of zero.) The nocopy_flag of the from_string helper is used for non-copying insertion of a string into an Any (in which the Any assumes ownership of the string). Unbounded strings can also be inserted into or extracted from an Any without the use of the from_string helper type by using the char* insertion and deletion operators.

The to_object helper type (see below) is used to extract an object reference from an Any as a generic CORBA::Object type. The Any extraction operator corresponding to the to_object helper type widens its contained object reference to CORBA::Object (if it contains one). No duplication of the object reference is performed by the to_object extraction operator.

In addition to the insertion/extraction operators defined in the Any class, corresponding to the basic IDL data types, the emitters generate global insertion/extraction operators for all types defined in IDL. This allows any type that can be defined in IDL to be inserted into or extracted from an Any in a type-safe manner.

In cases where the type-safe Any operators cannot be used, the Any non-default constructor (described above), the replace method, the type method, and the value method can be used to explicitly set or get the TypeCode and value contained by the Any.

Types

```
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 = 0);
    char *val;
    ULong bound;
    Boolean nocopy_flag;
};
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_object
{
    to_object(Object_ptr &obj) : ref(obj) {}
    Object_ptr &ref;
};
struct to_string
{
    to_string(char * &s, ULong b) : val(s), bound(b) {}
    char *&val;
    ULong bound;
};

```

Any::_nil

| | |
|-----------------------|---|
| Overview | Returns a nil CORBA::Any reference. |
| Original class | "CORBA::Any" on page 55 |

Intended Usage

This method is intended to be used by client and server applications to create a nil Any reference.

IDL Syntax

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

Return value

CORBA::Any_ptr

A nil Any reference.

Example

```

#include "corba.h"
...
CORBA::Any* any_ptr;
any_ptr = CORBA::Any::_nil();
...

```

Any::operator<<

| | |
|-----------------------|---|
| Overview | Inserts data into an Any. |
| Original class | "CORBA::Any" on page 55 |

Intended Usage

This operator is intended to be used for type-safe insertion of a data value into an Any. The C++ type of the data being inserted determines what TypeCode is automatically created and stored in the Any. The operators are type-safe in that they insure that an Any is not created with a TypeCode that doesn't match the value it holds.

When inserting a value into an Any, the previous value held by the Any is deallocated.

The `from_boolean`, `from_char`, and `from_octet` helper types are used to distinguish between the IDL types `boolean`, `char`, and `octet`, since these IDL types are mapped to the same C++ type. To insert a `boolean`, `octet`, or `char` into an Any, construct a helper object (by passing the data to be inserted to the helper's constructor), then use the helper object with the corresponding Any insertion operator.

The `from_string` helper type is used to insert a bounded or unbounded string into an Any, since both IDL types are mapped to `char*` in C++. (Unbounded strings are signified by constructing the `from_string` helper with a bound of zero.) The `nocopy_flag` of the `from_string` helper is used for non-copying insertion of a string into an Any (in which the Any assumes ownership of the string). Unbounded strings can also be inserted into an Any

without the use of the `from_string` helper type.

In addition to the insertion operators defined in the `Any` class, corresponding to the basic IDL data types, the emitters generate global insertion operators for all types defined in IDL. This allows any type that can be defined in IDL to be inserted into an `Any` in a type-safe manner. Both copying and non-copying insertion operators are defined in the bindings.

To insert an array into an `Any`, the `<array-name>_forany` helper type (defined in the emitted C++ bindings) should be used. In the C++ bindings, an array within a function argument list decays into a pointer to the first element, thus `Any`-insertion operators cannot be overloaded to distinguish between arrays of different sizes. Instead, `Any`-insertion operators are provided for each distinct `<array-name>_forany` type. To insert an array into an `Any`, create an appropriate `<array-name>_forany` object, initializing it from the array to be inserted, then use the global operator `<<` (the `Any` insertion operator) defined for that `<array-name>_forany` type. There is no `from_object` helper type corresponding to the `to_object` helper type.

IDL Syntax

```
void operator<<= (CORBA::Short data);
void operator<<= (CORBA::UShort data);
void operator<<= (CORBA::Long data);
void operator<<= (CORBA::ULong) data;
void operator<<= (CORBA::Float data);
void operator<<= (CORBA::Double data);
void operator<<= (const CORBA::Any &data);
void operator<<= (const char* data);
void operator<<= (const WChar* data);
void operator<<= (CORBA::Any::from_boolean data);
void operator<<= (CORBA::Any::from_char data);
void operator<<= (CORBA::Any::from_octet data);
void operator<<= (CORBA::Any::from_string data);
```

Parameters

data

The data to be inserted into the `Any`.

Example

```
#include "corba.h"
#include ...
/* Assert a value of short type properly insert or
 * extract from an Any
 */
CORBA::Any any;
CORBA::Short s1 = 1, s2 = 2;
/* s1 extracted from any */
any <<= s2 /* insert s2 into any */ any >>= s1
assert(s1 == s2);
/* Assert a value of the from/to helper type methods
 * which properly inserted or extracted
 */
CORBA::Any anyc;
char my_char = 'z';
CORBA::Char x_char = 'u';
/* insert my_char into anyc */
anyc <<= corba::any::from_char(my_char);
/* x_char extracted from anyc */
anyc >>= CORBA::Any::to_char(x_char);
assert(x_char == my_char);
```

Any::operator>>

| | |
|----------------|--|
| Overview | Extracts data from an <code>Any</code> . |
| Original class | "CORBA::Any" on page 55 |

Intended Usage

This operator is intended to be used for type-safe extraction of a data value from an `Any`. If the C++ type of the data being extracted matches the `TypeCode` in the `Any`, the operator's parameter is updated with the `Any`'s value. For simple types, the `Any`'s value is copied to the parameter passed to the extraction operator. Non-primitive types are extracted by pointer; if the extraction is successful, the pointer passed to the extraction operator is modified to point to the `Any`'s value. The `Any` retains ownership of the value and the caller must not delete it, and should not use the value after the `Any` is destroyed or given a new value. (For this reason, avoid using `Any` extraction operators to extract values into `<type>_var` variables.)

The `to_boolean`, `to_char`, and `to_octet` helper types are used to distinguish between the IDL types `boolean`, `char`, and `octet`, since these IDL types are mapped to the same C++ type. To extract a `boolean`, `octet`, or `char` from an `Any`, construct a helper object (by passing the data to be inserted to the helper's constructor), then use the helper object with the corresponding `Any` extraction operator.

The `to_string` helper type is used to extract a bounded or unbounded string from an `Any`, since both IDL types are mapped to `char*` in C++. (Unbounded strings are signified by constructing the `from_string` helper with a bound of zero.) Unbounded strings can also be extracted from an `Any` without the use of the `from_string` helper type.

In addition to the extraction operators defined in the `Any` class, corresponding to the basic IDL data types, the emitters generate global extraction operators for all types defined in IDL. This allows any type that can be defined in IDL to be extracted from an `Any` in a type-safe manner.

To extract an array from an `Any`, the `<array-name>_forany` helper type (defined in the emitted C++ bindings) should be used. In the C++ bindings, an array within a function argument list decays into a pointer to the first element, thus `Any`-extraction operators cannot be overloaded to distinguish between arrays of different sizes. Instead, `Any`-extraction operators are provided for each distinct `<array-name>_forany` type. To extract an array from an `Any`, create an appropriate `<array-name>_forany` object, initializing it from the array to be extracted, then use the global operator `>>` (the `Any` extraction operator) defined for that `<array-name>_forany` type.

After extracting a bounded string or an array from an `Any`, applications are responsible for checking the `Any`'s `TypeCode` (using the `Any::type()` method) to insure they do not overstep the bounds of the array or string when using the extracted value.

The `to_object` helper type is used to extract an object reference from an `Any` as a generic `CORBA::Object` type. The `Any` extraction operator corresponding to the `to_object` helper type widens its contained object reference to `CORBA::Object` (if it contains one). No duplication of the object reference is performed by the `to_object` extraction operator.

IDL Syntax

```
CORBA::Boolean operator>>= (CORBA::Short& data) const;
CORBA::Boolean operator>>= (CORBA::UShort& data) const;
CORBA::Boolean operator>>= (CORBA::Long& data) const;
CORBA::Boolean operator>>= (CORBA::ULong& data) const;
CORBA::Boolean operator>>= (CORBA::Float& data) const;
CORBA::Boolean operator>>= (CORBA::Double& data) const;
CORBA::Boolean operator>>= (CORBA::Any& data) const;
CORBA::Boolean operator>>= (char*& data) const;
CORBA::Boolean operator>>= (wchar_t*& data) const;
CORBA::Boolean operator>>= (CORBA::Any::to_boolean data) const;
CORBA::Boolean operator>>= (CORBA::Any::to_char data) const;
CORBA::Boolean operator>>= (CORBA::Any::to_octet data) const;
CORBA::Boolean operator>>= (CORBA::Any::to_object data) const;
CORBA::Boolean operator>>= (CORBA::Any::to_string data) const;
```

Parameters

data

The data whose value is to be extracted from the `Any`.

Return value

CORBA::Boolean

A non-zero result indicates successful extraction and that the `Any` actually contained the type of data requested. A zero return value indicates that the `Any`'s `TypeCode` does not match the C++ type of the operator's parameter and that nothing was extracted. For primitive types, a zero return value indicates that the parameter has not been modified. For non-primitive types, a zero return value indicates that the pointer passed to the operator has been set to `NULL`.

Example

See example in [“Any::operator<<” on page 57](#).

Any::replace

| | |
|----------------|--|
| Overview | Replaces the data value and TypeCode held by an Any. |
| Original class | "CORBA::Any" on page 55 |

Intended Usage

This method is intended to be used to reinitialize an Any to hold a new TypeCode and data value. This method is not type-safe (no checking is done to insure that the given TypeCode actually matches the given data value.) This method is intended to be used only when the type-safe Any insertion operators cannot be used.

The replace interface mirrors the interface to Any's non-default constructor.

If the Any previously owned the value it contained, that value is deleted and the new value is stored in the Any. The TypeCode previously contained by the Any is released, and the input TypeCode is duplicated and stored in the Any. If the release parameter is nonzero, then the Any assumes ownership of the new value, and the application should make no assumptions about the continued lifetime of the value.

IDL Syntax

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

Parameters

tc

The TypeCode describing the value parameter. The Any duplicates this TypeCode, so the caller retained ownership of the input TypeCode.

value

The new data to be stored in the Any. The type of this data must be described by the tc parameter. This parameter can be NULL.

If the value is a simple type (char, octet, float, etc.), the *value* parameter should be a pointer to the data. If the value is a string, the *value* parameter should be of type char**. If the value is an object corresponding to an IDL interface (such as MyInterface), the *value* parameter should be of type MyInterface_ptr. If the value is a TypeCode, the *value* parameter should be of type CORBA::TypeCode_ptr. If the value is a constructed IDL type (struct, sequence, union), the value parameter should be a pointer to the data. If the value is an Any, the *value* parameter should be of type CORBA::Any*. If the value is an IDL array, the *value* parameter should be a pointer to the first element of the array.

release

Specifies whether the Any should assume ownership of the value parameter (whether the value should be deallocated when the Any is released). Default is zero (meaning that the Any does not assume ownership of the value).

Example

```
#include "corba.h"  
...  
CORBA::Any any;  
const Val7 = 7;  
CORBA::ULong ul_val = Val7;  
/* release == > any owns value memory */  
CORBA::Boolean any_owns_p = 0;  
/* put a Ulong into any */  
any.replace(CORBA::_tc_ulong, (void*) &ul_val, any_owns_p);  
...
```

Any::type

| | |
|-----------------------|--|
| Overview | Accesses the TypeCode contained by an Any. |
| Original class | "CORBA::Any" on page 55 |

Intended Usage

This method is intended to be used to access the TypeCode associated with an Any (the TypeCode that describes the data held by the Any). The caller must subsequently release the TypeCode using `CORBA::release(TypeCode_ptr)`.

In many cases, Any objects can be used without explicit manipulation of TypeCodes, using the type-safe insertion/extraction operators defined for Any. The `type()` method is for situations in which the type-safe Any interface is not applicable, or to determine the type of variable needed for extraction.

IDL Syntax

```
TypeCode_ptr type() const;
```

Return value

CORBA::TypeCode_ptr

The TypeCode contained by the Any. The caller must subsequently release the TypeCode using `CORBA::release(TypeCode_ptr)`.

Example

```
#include "corba.h"
...
CORBA::TypeCode_ptr tcp;
CORBA::Any constAnyVar6;
constAnyVar6 <<= (corba::short)(6);
tcp="constAnyVar6.type()";
...
```

CORBA module: ArrayDef Interface

| | |
|-----------------------------|---|
| Overview | An ArrayDef represents an OMG IDL array type. |
| File name | somir.idl |
| Local-only | True |
| Ancestor interfaces | "IDLType Interface" on page 127 |
| Exceptions | "CORBA::SystemException" on page |
| Supported operations | "ArrayDef::element_type" on page 62 |
| | "ArrayDef::element_type_def" on page 62 |
| | "ArrayDef::length" on page 63 |
| | "IDLType::type" on page 127 |

Intended Usage

The ArrayDef interface is used by the Interface Repository to represent an OMG IDL array data type. The ArrayDef is not a named Interface Repository data type (it is in a group of interfaces known as Anonymous types). An ArrayDef may be created using the `create_array` operation of the Repository interface, by specifying the length of the array and a `CORBA::IDLType*` indicating the array element type.

Since an ArrayDef object only represents a single dimension of an array, multi-dimensional IDL arrays are represented by multiple ArrayDef objects, one per array dimension. The `element_type_def` attribute of the ArrayDef representing the index that is on the far left side of the array, as defined in IDL, refers to the ArrayDef representing the next index to the right, and so on. The innermost ArrayDef represents the rightmost index and the element type of the multi-dimensional OMG IDL array.

IDL syntax

```
module CORBA
{
    interface ArrayDef:IDLType
    {
        attribute unsigned long length;
        readonlyattribute TypeCode element_type;
        attribute IDLType element_type_def;
    };
};
```

ArrayDef::element_type

| | |
|---------------------------|---|
| Overview | The element_type operation returns a type (CORBA::TypeCode *) representative of the array element of an ArrayDef. |
| Original interface | "ArrayDef Interface" on page 61 |
| Exceptions | "CORBA::SystemException" on page |

Intended Usage

The element_type attribute of an ArrayDef object points to a CORBA::TypeCode that represents the type of the array element. The element_type read operation returns a copy of the CORBA::TypeCode referenced by the element_type attribute.

IDL Syntax

```
readonly attribute TypeCode element_type;
```

Input parameters

None.

Return values

TypeCode *

The returned value is a pointer to a copy of the CORBA::TypeCode referenced by the element_type attribute. The memory is owned by the caller and can be returned by invoking CORBA::release.

Example

```
// C++
// assume that 'this_array' has already been initialized
CORBA::ArrayDef * this_array;
// retrieve the TypeCode which represents the array element
CORBA::TypeCode * array_element_type;
array_element_type = this_array-> element_type ();
```

ArrayDef::element_type_def

| | |
|---------------------------|--|
| Overview | The element_type_def read and write operations allow the access and update of the element type definition of an array definition (ArrayDef) in the Interface Repository. |
| Original interface | "ArrayDef Interface" on page 61 |
| Exceptions | "CORBA::SystemException" on page |

Intended Usage

The type of the elements within an array definition is identified by the element_type_def attribute.

IDL Syntax

```
attribute IDLType element_type_def;
```

Read operations

Input parameters

none

Return values

CORBA::IDLType_ptr

The returned object is a pointer to a copy of the IDLType referenced by the element_type_def attribute of the ArrayDef object. The returned object is owned by the caller and can be released by invoking CORBA::release.

Write operations

Input parameters

CORBA::IDLType_ptr element_type_def

The element_type_def parameter represents the new element definition for the ArrayDef.

Return values

none

Example

```
// C++
// assume that 'this_array' and 'this_union' have already been initialized
CORBA::ArrayDef * this_array;
CORBA::UnionDef * this_union;
// change the array element type definition to 'this_union'
this_array-> element_type_def (this_union);
// read the element type definition from 'this_array'
CORBA::IDLType * returned_element_type_def;
returned_element_type_def = this_array-> element_type_def ();
```

ArrayDef::length

| | |
|--------------------|--|
| Overview | The length read and write operations allow the access and update of the length attribute of an array definition (CORBA::ArrayDef) within the Interface Repository. |
| Original interface | "ArrayDef Interface" on page 61 |
| Exceptions | "CORBA::SystemException" on page |

Intended Usage

The length attribute specifies the number of elements in the array. Read and write length operations are supported.

IDL Syntax

```
attribute unsigned long length;
```

Read operations

Input parameters

none

Return values

CORBA::ULong

The returned value is the current value of the length attribute of the array definition (CORBA::ArrayDef) object.

Write operations

Input parameters

CORBA::ULong length

The length parameter is the new value to which the length attribute of the CORBA::ArrayDef object will be set.

Return values

none

Example

```
// C++
// assume that 'this_array' has already been initialized
CORBA::ArrayDef * this_array;
// change the length attribute of the array definition
CORBA::ULong new_length = 51;
this_array-> length (new_length);
// obtain the length of an array definition
CORBA::ULong returned_length;
returned_length = this_array-> length ();
```

CORBA module: AttributeDef Interface

| | |
|-----------------------------|--|
| Overview | The AttributeDef interface is used within the Interface Repository to represent the information that defines an attribute of an interface. |
| File name | somir.idl |
| Local-only | True |
| Ancestor interfaces | "Contained Interface" on page 79 |
| Exceptions | "CORBA::SystemException" on page 64 |
| Supported operations | "AttributeDef::describe" on page 64 |
| | "AttributeDef::mode" on page 65 |
| | "IDLType::type" on page 127 |
| | "AttributeDef::type_def" on page 66 |

Intended Usage

The AttributeDef object is used to represent the information that defines an attribute of an interface. An AttributeDef may be created by calling the create_attribute operation of the InterfaceDef interface. The create_attribute parameters include the unique RepositoryId (CORBA::RepositoryId), the name (CORBA::Identifier), the version (CORBA::VersionSpec), the type (CORBA::IDLType*) to indicate the type of the attribute, and a parameter to indicate the mode of the attribute (read, read/write, etc.).

IDL syntax

```
module CORBA
{
    enum AttributeMode {ATTR_NORMAL, ATTR_READONLY};
    interface AttributeDef:Contained
    {
        readonlyattribute TypeCodetype;
        attribute IDLType type_def;
        attribute AttributeMode mode;
    };
    struct AttributeDescription
    {
        Identifier name;
        RepositoryId id;
        RepositoryId defined_in;
        VersionSpec version;
        TypeCode type;
        AttributeMode mode;
    };
};
```

AttributeDef::describe

| | |
|---------------------------|--|
| Overview | The describe operation returns a structure containing information about a CORBA::AttributeDef Interface Repository object. |
| Original interface | "CORBA module: AttributeDef Interface" on page 64 |
| Exceptions | "CORBA::SystemException" on page 64 |

Intended Usage

The inherited describe operation returns a structure (CORBA::Contained::Description) that

contains information about a CORBA::AttributeDef Interface Repository object. The CORBA::Contained::Description structure has two fields: kind (CORBA::DefinitionKind data type), and value (CORBA::Any data type).

The kind of definition described by the returned structure is provided using the kind field, and the value field is a CORBA::Any that contains the description that is specific to the kind of object described. When the describe operation is invoked on an attribute (CORBA::AttributeDef) object, the kind field is equal to CORBA::dk_Attribute and the value field contains the CORBA::AttributeDescription structure.

IDL Syntax

```

struct AttributeDescription
{
    Identifier name;
    RepositoryId id;
    RepositoryId defined_in;
    VersionSpec version;
    TypeCode type;
    AttributeMode mode;
};
struct Description
{
    DefinitionKind kind;
    any value;
};
Description describe ();

```

Input parameters

None.

Return values

Description *

The returned value is a pointer to a CORBA::Contained::Description structure. The memory is owned by the caller and can be removed by invoking delete.

Example

```

// C++
// assume that 'this_attribute' has already been initialized
CORBA::AttributeDef * this_attribute;
// retrieve a description of the attribute
CORBA::AttributeDef::Description * returned_description;
returned_description = this_attribute-> describe ();
// retrieve the attribute description from the returned description
// structure
CORBA::AttributeDescription * attribute_description;
attribute_description = (CORBA::AttributeDescription *)
returned_description-> value.value ();

```

AttributeDef::mode

| | |
|---------------------------|--|
| Overview | The mode read and write operations allow the access and update of the mode attribute of an attribute definition (CORBA::AttributeDef) within the Interface Repository. |
| Original interface | "CORBA module: AttributeDef Interface" on page 64 |
| Exceptions | "CORBA::SystemException" on page |

Intended Usage

The mode attribute specifies read only or read/write access for this attribute. Read and write mode operations are supported with parameters as defined below.

IDL Syntax

```

attribute AttributeMode mode;

```

Read operations

Input parameters

none

Return values

CORBA::AttributeMode mode

The returned value is the current value of the mode attribute of the attribute definition (CORBA::AttributeDef) object.

Write operations

Input parameters

CORBA::AttributeMode mode

The mode parameter is the new value to which the mode attribute of the CORBA::AttributeDef object will be set. Valid mode values include CORBA::ATTR_NORMAL (read/write access) and CORBA::ATTR_READONLY (read only access).

Return values

none

Example

```
// C++
// assume that 'this_attribute' has already been initialized
CORBA::AttributeDef * this_attribute;
// set the new mode in the attribute definition
CORBA::AttributeMode new_mode = CORBA::ATTR_READONLY;
this_attribute-> mode (new_mode);
// retrieve the mode from the attribute definition
CORBA::AttributeMode returned_mode;
returned_mode = this_attribute-> mode ();
```

AttributeDef::type_def

| | |
|---------------------------|---|
| Overview | The type_def operation returns a pointer to an IDLType that is representative of the type of the attribute defined by the AttributeDef. |
| Original interface | "CORBA module: AttributeDef Interface" on page 64 |
| Exceptions | "CORBA::SystemException" on page |

Intended Usage

The type_def attribute within an AttributeDef references an IDLType that identifies the type of attribute. Both read and write type_def operations are supported, the parameters of which are identified below.

IDL Syntax

```
attribute IDLType type_def;
```

Read operations

Input parameters

none

Return values

CORBA::IDLType_ptr

The returned CORBA::IDLType * is a pointer to a copy of the information referenced by the type_def attribute. The object and the associated memory are owned by the caller and can be released by invoking CORBA::release.

Write operations

Input parameters

CORBA::IDLType_ptr type_def

The type_def input parameter identifies the new setting for the type_def attribute.

Return values

none

Example

```
// C++
// assume that 'this_attribute' and 'pk_long_def'
// have already been initialized
CORBA::AttributeDef * this_attribute;
CORBA::PrimitiveDef * pk_long_def;
// set the type_def attribute of the AttributeDef
// to represent a CORBA::Long
this_attribute-> type_def (pk_long_def);
// retrieve the type_def attribute from the AttributeDef
CORBA::IDLType * attributes_type_def;
attributes_type_def = this_attribute-> type_def();
```

CORBA module: BOA Class

| | |
|--|--|
| Overview | Provides services for writing server applications. |
| File name | boa.h |
| Nested classes | CORBA::BOA::DynamicImplementation |
| Supported methods | "BOA::_duplicate" on page 67 |
| | "BOA::_nil" on page 68 |
| | "BOA::create" on page 68 |
| | "BOA::deactivate_impl" on page 69 |
| | "BOA::dispose" on page 70 |
| | "BOA::execute_next_request" on page 71 |
| | "BOA::execute_request_loop" on page 71 |
| | "BOA::get_id" on page 72 |
| | "BOA::get_principal" on page 73 |
| | "BOA::impl_is_ready" on page 73 |
| | "BOA::request_pending" on page 74 |
| Methods introduced by BOA in the CORBA specification but not implemented in this product. 1 | change_implementation |
| | deactivate_obj |
| | obj_is_ready |

Intended Usage

The BOA (Basic Object Adapter) class is intended to be used by application-specific server programs, to access server-side services of the ORB. The BOA class provides methods for activating and deactivating the server and executing remote requests from client applications. The BOA class also provides methods that allow the server application to participate in the exporting and importing of object references and the selection of threads on which remote requests are dispatched. Most of the BOA methods are intended to be called from an application-specific server program. The BOA::get_principal method, however, is typically called from an implementation of an IDL interface residing in a server, to determine the identity of the remote caller.

Types

```
typedef CORBA::ReferenceData * SOMLINK
    somdTD_obj_to_refdata (CORBA::Object_ptr obj);
typedef CORBA::Object_ptr SOMLINK
    somdTD_refdata_to_obj (CORBA::ReferenceData *refdata);
typedef void SOMLINK somdTD_thread_dispatch (CORBA::Request_ptr req);
```

Constants

```
static const CORBA::Flags SOMD_WAIT;
static const CORBA::Flags SOMD_NO_WAIT;
```

BOA::_duplicate

1 If invoked, a CORBA::SystemException is thrown

| | |
|-----------------------|--|
| Overview | Duplicates a BOA object. |
| Original class | "CORBA module: BOA Class" on page 67 |

Intended Usage

This method is intended to be used by client and server applications to duplicate a reference to a BOA object.

IDL Syntax

```
static CORBA::BOA_ptr _duplicate (CORBA::BOA_ptr p);
```

Input parameters

p

The BOA object to be duplicated. The reference can be nil, in which case the return value will also be nil.

Return values

CORBA::BOA_ptr

The new BOA object reference.

Example

See example in ["CORBA::Object::_duplicate" on page 153](#)

BOA::_nil

| | |
|-----------------------|--|
| Overview | Returns a nil CORBA::BOA reference. |
| Original class | "CORBA module: BOA Class" on page 67 |

Intended Usage

This method is intended to be used by client and server applications to create a nil BOA reference.

IDL Syntax

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

Return value

CORBA::BOA_ptr

A nil BOA reference.

Example

See example in ["CORBA::Object::_nil" on page 157](#)

BOA::create

| | |
|-----------------------|--|
| Overview | Maps ReferenceData to a local object, and prepares that object for export. |
| Original class | "CORBA module: BOA Class" on page 67 |
| Exceptions | "CORBA::SystemException" on page |

Intended Usage

Typical server applications need never use this method.

This method is part of the CORBA specification.

IDL Syntax

```
virtual CORBA::Object_ptr create
```

```
(const CORBA::ReferenceData& refdata,
CORBA::InterfaceDef_ptr intf,
CORBA::ImplementationDef_ptr impldef);
```

Input parameters

refdata

The application-specific ReferenceData of an object residing in a server.

intf

The InterfaceDef object, retrieved from the Interface Repository, that describes the interface supported by the object described by the refdata parameter. Currently, this parameter is unused and can be NULL. The caller retains ownership of this object .

impldef

The ImplementationDef of the server in which the call is being made. Currently, this parameter is unused and can be NULL. The caller retains ownership of this object.

Return value

CORBA::Object_ptr

The local object in the server that corresponds to the input ReferenceData, after it has been prepared for export. Ownership of this object reference is transferred to the caller, and should be subsequently released using CORBA::release.

Example

```
#include "corba.h"
extern CORBA::BOA_ptr srvboa; /* assume previously initialized
                               using CORBA::ORB::BOA_init */
...
::CORBA::ReferenceData * rd = (::CORBA::ReferenceData *) NULL;
rd = srvboa->get_id(this);
::CORBA::Object_ptr objPtr =
    srvboa->create(*rd,
                  CORBA::InterfaceDef::_nil(),
                  CORBA::ImplementationDef::_nil());
...
```

BOA::deactivate_impl

| | |
|-----------------------|---|
| Overview | Causes a server to stop accepting incoming request messages and informs the somorbd daemon that it is no longer active. |
| Original class | "CORBA module: BOA Class" on page 67 |
| Exceptions | "CORBA::SystemException" on page |

Intended Usage

This method is intended to be used by every server application, to indicate that it no longer wishes to accept incoming request messages from remote clients, prior to server termination (whether normal or abnormal). It should only be invoked if the server has previously successfully called BOA::impl_is_ready.

The method informs the somorbd daemon that the server is no longer active, and that subsequent requests to locate that logical server should cause a new instance of the server to be automatically activated. It also prevents any new request messages from being received by the server; clients issuing such requests will receive SystemExceptions indicating a communications failure. Any requests received by the server prior to calling BOA::deactivate_impl that have not yet been serviced (by a call to BOA::execute_request_loop or BOA::execute_next_request) will be deleted; no response will be sent to the clients that sent them.

This method is part of the CORBA specification.

IDL Syntax

```
virtual void deactivate_impl (
CORBA::ImplementationDef_ptr impldef);
```

Parameters

impldef

The ImplementationDef of the server making the call. This should be the same ImplementationDef originally passed to BOA::impl_is_ready. The caller retains ownership of this parameter.

Example

```
#include "corba.h"
void main(int argc, char* argv[])
{
    /* Initialize the server's ImplementationDef, ORB, and BOA: */
    CORBA::ImplRepository_ptr implrep = new CORBA::ImplRepository;
    /* assume dummyServer is already registered in
    the implementation repository */
    CORBA::ImplementationDef_ptr imp =
        implrep->find_impldef_by_alias ("dummyServer");
    /* Assume op-param is initialized. For workstation initialize to "DSOM" */
    char * op-param;
    /* Assume bp-param is initialized. For workstation initialize to
    "DSOM_BOA" */
    char * bp-param;
    static CORBA::ORB_ptr op = CORBA::ORB_init(argc, argv, op-param);
    static CORBA::BOA_ptr bp = op->BOA_init(argc, argv, bp-param);
    bp->impl_is_ready(imp);
    ...
    bp->deactivate_impl(imp);
    ...
}
```

BOA::dispose

| | |
|----------------|--|
| Overview | Destroys an object residing in a server. |
| Original class | "CORBA module: BOA Class" on page 67 |

Intended Usage

This method can be used to destroy (delete) a local object residing in a server. All outstanding references to the object are henceforth invalid. Outstanding remote references (proxies) to the object are valid only if the server is capable of reactivating to the object. The current implementation of this method simply deletes the input object.

This method is part of the CORBA specification.

IDL Syntax

```
virtual void dispose(CORBA::Object_ptr obj);
```

Parameters

obj

The object to be deleted.

Example

```
#include "corba.h"
void main(int argc, char* argv[])
{
    /* Initialize the server's ImplementationDef, ORB, and BOA: */
    CORBA::ImplRepository_ptr implrep = new CORBA::ImplRepository;
    /* Assume dummyServer is already registered in
    the implementation repository */
    CORBA::ImplementationDef_ptr imp =
        implrep->find_impldef_by_alias ("dummyServer");
    extern static CORBA::ORB_ptr op; /* assume previously initialized */
    extern static CORBA::BOA_ptr bp; /* assume previously initialized */
    bp->impl_is_ready(imp);
    ...
    /* Assume that p is a local object pointer already declared
    and defined */
    bp->dispose(p);
    ...
}
```

```
#include "corba.h"
void main(int argc, char* argv[])
{
    /* Initialize the server's ImplementationDef, ORB, and BOA: */
    CORBA::ImplRepository_ptr implrep = new CORBA::ImplRepository;
    /* Assume dummyServer is already registered in
    the implementation repository */
    CORBA::ImplementationDef_ptr imp =
        implrep->find_impldef_by_alias ("dummyServer");
    static CORBA::ORB_ptr op = CORBA::ORB_init(argc, argv, "DSOM");
    static CORBA::BOA_ptr bp = op->BOA_init(argc, argv, "DSOM_BOA");
    bp->impl_is_ready(imp);
    ...
    /* Assume that p is a local object pointer already declared
```

```

        and defined */
        bp->dispose(p);
        ...
    }

```

BOA::execute_next_request

| | |
|----------------|---|
| Overview | Executes the next pending remote request in a server application. |
| Original class | "CORBA module: BOA Class" on page 67 |
| Exceptions | "CORBA::SystemException" on page |

Intended Usage

This method is intended to be used by a server application to execute the next remote request received from a remote client, and send the response to the waiting client. Both blocking and non-blocking calls are supported. Requests are executed in first-in-first-out order only. This method should be called only after CORBA::BOA::impl_is_ready has been called successfully.

This method is an IBM^(R) extension to the CORBA specification.

IDL Syntax

```
virtual CORBA::Status execute_next_request (CORBA::Flags waitFlag);
```

Input parameters

waitFlag

Whether the application wants to wait (block), if there is no request currently available to process. Valid values are CORBA::BOA::SOMD_WAIT and CORBA::BOA::SOMD_NO_WAIT.

Return values

CORBA::Status

A zero return value indicates success. If the input parameter is CORBA::BOA::SOMD_NO_WAIT, a return value of SOMDERROR_NoMessages indicates that there is no available request to service.

Example

```

#include "corba.h"
void main(int argc, char* argv[])
{
    /* Initialize the server's ImplementationDef, ORB, and BOA: */
    CORBA::ImplRepository_ptr implrep = new CORBA::ImplRepository;
    /* Assume dummyServer is already registered in
       the implementation repository */
    CORBA::ImplementationDef_ptr imp =
        implrep->find_impldef_by_alias ("dummyServer");
    extern static CORBA::ORB_ptr op; /* assume previously initialized */
    extern static CORBA::BOA_ptr bp; /* assume previously initialized */
    bp->impl_is_ready(imp);
    ...
    /* Execute the next pending remote request */
    while(1)
        bp->execute_next_request(CORBA::BOA::SOMD_WAIT);
    ...
}

```

```

#include "corba.h"
void main(int argc, char* argv[])
{
    /* Initialize the server's ImplementationDef, ORB, and BOA: */
    CORBA::ImplRepository_ptr implrep = new CORBA::ImplRepository;
    /* Assume dummyServer is already registered in
       the implementation repository */
    CORBA::ImplementationDef_ptr imp =
        implrep->find_impldef_by_alias ("dummyServer");
    static CORBA::ORB_ptr op = CORBA::ORB_init(argc, argv, "DSOM");
    static CORBA::BOA_ptr bp = op->BOA_init(argc, argv, "DSOM_BOA");
    bp->impl_is_ready(imp);
    ...
    /* Execute the next pending remote request */
    while(1)
        bp->execute_next_request(CORBA::BOA::SOMD_WAIT);
    ...
}

```

BOA::execute_request_loop

| | |
|-----------------------|--|
| Overview | Repeatedly executes remote requests in a server application. |
| Original class | "CORBA module: BOA Class" on page 67 |
| Exceptions | "CORBA::SystemException" on page |

Intended Usage

This method is intended to be used by a server application to repeatedly execute remote requests as they are received from remote clients, and sends the responses to the waiting clients. Both blocking and non-blocking calls are supported. Requests are executed in first-in-first-out order only, by calling CORBA::BOA::execute_next_request. This method should be called only after CORBA::BOA::impl_is_ready has been called successfully.

This method is an IBM extension to the CORBA specification.

IDL Syntax

```
virtual CORBA::Status execute_request_loop (CORBA::Flags waitFlag);
```

Input parameters

waitFlag

Whether the application wants to wait (block), when there are no more requests available to process. Valid values are CORBA::BOA::SOMD_WAIT and CORBA::BOA::SOMD_NO_WAIT.

Return values

CORBA::Status

If the input parameter is CORBA::BOA::SOMD_NO_WAIT, SOMDERROR_NoMessages is returned when there are no more available requests to service. Otherwise, this method never returns to the caller.

Example

See example in ["CORBA::ORB::BOA_init" on page 169](#)

BOA::get_id

| | |
|-----------------------|---|
| Overview | Returns the ReferenceData associated with a local object in a server. |
| Original class | "CORBA module: BOA Class" on page 67 |
| Exceptions | "CORBA::SystemException" on page |

Intended Usage

This method is intended to be used by a server application, to access the ReferenceData used by that server to identify the object.

Typical server applications would only need to call this method to obtain the ReferenceData required by the CORBA::BOA::create method.

This method is part of the CORBA specification.

IDL Syntax

```
virtual CORBA::ReferenceData *get_id (CORBA::Object_ptr obj);
```

Input parameters

obj

The local object for which ReferenceData is needed. If this parameter is NULL or is a proxy object (rather than a local object in a server), an exception is thrown.

Return values

CORBA::ReferenceData*

The ReferenceData associated with the given object. Ownership of the ReferenceData is transferred to the caller.

Example

See example in [“CORBA::BOA::create” on page 68](#) .

BOA::get_principal

| | |
|----------------|--|
| Overview | Returns a Principal object identifying, in a server, the client of a remote request. |
| Original class | “CORBA module: BOA Class” on page 67 |
| Exceptions | “CORBA::SystemException” on page |

Intended Usage

This method is intended to be used by implementations of IDL interfaces, residing in a server process, to find the identity of the calling client. This might be used, for example, to implement security authorization checks.

This method is part of the CORBA specification.

IDL Syntax

```
virtual CORBA::Principal_ptr get_principal (  
    CORBA::Object_ptr obj,  
    CORBA::Environment_ptr env);
```

Input parameters

obj

The target of a remote invocation in a server. Typically, it is the "this" pointer in C++ for the method calling CORBA::BOA::get_principal. Currently this parameter is not used and NULL can be passed.

env

Currently unused (NULL can be passed).

Return values

CORBA::Principal_ptr

A Principal object identifying the client that initiated the remote request. If CORBA::BOA::get_principal is called outside the context of any remote request, NULL is returned. The caller does not assume ownership of the returned Principal object and should not delete it.

Example

```
#include "corba.h"  
/* Assume previously initialized using CORBA::ORB::BOA_init () */  
extern CORBA::BOA_ptr srvboa;  
...  
::CORBA::Boolean tmpBoolean;  
::CORBA::Principal_ptr prncpl_ptr;  
/* Assume the following is called from within an implementation  
of some IDL operation */  
prncpl_ptr = srvboa->get_principal(this,  
                                CORBA::Environment::_nil());  
tmpBoolean = ::CORBA::is_nil(prncpl_ptr);  
/* Error checking */  
...
```

BOA::impl_is_ready

| | |
|----------------|---|
| Overview | Initializes an application as a server, allows it to accept incoming request messages, and registers it with the somorbd daemon |
| Original class | “CORBA module: BOA Class” on page 67 |

Intended Usage

This method is intended to be used by all server applications, to initialize themselves. A server application cannot receive remote requests and cannot export objects (for instance, using `CORBA::ORB::object_to_string` or `CORBA::BOA::create`) without first calling `CORBA::BOA::impl_is_ready`. This method initializes the server's communications resources so that it can accept incoming request messages, and (optionally) registers the server with the `somorbd` daemon so that client applications can locate it via the daemon.

After a server has called `CORBA::BOA::impl_is_ready`, it should call `CORBA::BOA::deactivate_impl` before termination (either normal or abnormal), to inform the `somorbd` daemon that it is no longer active.

This method is part of the CORBA specification.

IDL Syntax

```
virtual void impl_is_ready(CORBA::ImplementationDef_ptr impldef,
                          CORBA::Boolean registration = 1);
```

Parameters

impldef

The `ImplementationDef`, obtained from the Implementation Repository, that describes the server. The `ImplementationDef` is typically obtained using the `CORBA::ImplRepository find_impldef` or `find_impldef_by_alias` method. On-the-fly servers that are not registered in the Implementation Repository can create `ImplementationDef` objects using operator `new`.

registration

The default value (1) indicates that the server should register itself with the `somorbd` daemon. A zero value indicates that the server should not register itself with the `somorbd` daemon; this should only be done for lightweight servers of transient objects. This parameter is an IBM extension to the CORBA specification.

Example

See example in ["CORBA::ORB::BOA_init" on page 169](#)

BOA::request_pending

| | |
|----------------|---|
| Overview | Determines whether there are any requests in a server waiting to be serviced. |
| Original class | "CORBA module: BOA Class" on page 67 |

Intended Usage

This method is intended to be used by a server application to determine whether there are any outstanding requests (from remote clients) waiting to be serviced. Hence, this method can be used to determine whether a blocking call to `CORBA::BOA::execute_next_request` will block.

This call does not modify the queue of waiting requests.

This method is an IBM extension to the CORBA specification.

IDL Syntax

```
virtual CORBA::Boolean request_pending ();
```

Return values

CORBA::Boolean

A zero return value indicates that there are no outstanding requests waiting to be processed. A one return value indicates that there is at least one request pending.

Example

```
#include "corba.h"
void main(int argc, char* argv[])
{
    /* Initialize the server's ImplementationDef, ORB, and BOA: */
    CORBA::ImplRepository_ptr implrep = new CORBA::ImplRepository;
    /* Assume dummyServer is already registered in the
       implementation repository */
    CORBA::ImplementationDef_ptr imp =
        implrep->find_impldef_by_alias ("dummyServer");
    static CORBA::ORB_ptr op = CORBA::ORB_init(argc, argv, "DSOM");
    static CORBA::BOA_ptr bp = op->BOA_init(argc, argv, "DSOM_BOA");
    bp->impl_is_ready(imp);
    ...
    /* Determine if there's request waiting */
    CORBA::Boolean retval = bp->request_pending();
    ...
    /* Stop processing requests */
    bp->interrupt_server();
    ...
}
```

```
#include "corba.h"
void main(int argc, char* argv[])
{
    /* Initialize the server's ImplementationDef, ORB, and BOA: */
    CORBA::ImplRepository_ptr implrep = new CORBA::ImplRepository;
    /* Assume dummyServer is already registered in the
       implementation repository */
    CORBA::ImplementationDef_ptr imp =
        implrep->find_impldef_by_alias ("dummyServer");
    extern static CORBA::ORB_ptr op; /* assume previously initialized */
    extern static CORBA::BOA_ptr bp; /* assume previously initialized */
    bp->impl_is_ready(imp);
    ...
    /* Determine if there's request waiting */
    CORBA::Boolean retval = bp->request_pending();
    ...
    /* Stop processing requests */
    bp->interrupt_server();
    ...
}
```

CORBA module: BOA::DynamicImplementation Class

| | |
|--------------------------|---|
| Overview | Allows an object to be dynamically dispatched in a server. |
| File name | boa.h |
| Supported methods | "BOA::DynamicImplementation::invoke" on page 75 |

Intended Usage

This class is intended to be subclassed by implementations of IDL interfaces so that those implementations can be invoked dynamically in a server that was not statically linked with the C++ bindings for that IDL interface. For example, this technique might be used to implement objects residing in an inter-ORB bridge server, or gateway. This class is part of the Dynamic Skeleton Interface (DSI).

Subclasses of CORBA::BOA::DynamicImplementation must implement the invoke method.

BOA::DynamicImplementation::invoke

| | |
|-----------------------|--|
| Overview | Invokes a method dynamically within a server. |
| Original class | "CORBA::BOA::DynamicImplementation" on page 75 |
| Exceptions | This method must not throw any exceptions. Instead, exceptions should be stored on the input ServerRequest object, using the exception method. |

Intended Usage

This pure virtual method is intended to be overridden in subclasses of CORBA::BOA::DynamicImplementation. It is never called directly by applications; rather, the

BOA residing in a server calls this method to dispatch remote calls on objects that inherit from CORBA::BOA::DynamicImplementation. This method is part of the Dynamic Skeleton Interface (DSI), used primarily to construct inter-ORB bridges or gateway servers.

When a remote invocation arrives at a server, if the target of the invocation is an object that inherits from CORBA::BOA::DynamicImplementation, BOA calls the invoke method on the target object. BOA constructs and passes in a ServerRequest object that contains all the information about the incoming request that is needed for the object to dispatch it. As an example, an implementation of the invoke method could do the following:

- Obtain the name of the operation to be invoked from the ServerRequest.
- Discover the signature of the operation to be invoked (for example, using the Interface Repository or a cache of InterfaceDef objects, or by invoking `_get_interface` on the target object).
- Create an NVList containing the TypeCodes (but not the values) corresponding to the signature of the operation to be invoked.
- Call `ServerRequest::params`, passing in the NVList; the `ServerRequest::params` method stores the in and inout parameter values in the NVList.
- Dispatch the operation on the target object using the in and inout parameter values now available from the NVList.
- Store the inout and out parameter values in the NVList.
- Call `ServerRequest::result` to record the operation result.

The BOA then sends the response to the calling client. If an exception is thrown by the dispatched operation, the invoke method must catch it and record it by calling `ServerRequest::exception`. `CORBA::BOA::DynamicImplementation::invoke` must never throw any exceptions.

IDL Syntax

```
virtual void invoke (ServerRequest_ptr request,
                   Environment &env) throw () = 0;
```

Input parameters

request

A ServerRequest object that provides information about the operation to be invoked, the target object, and the values of the in and inout parameters.

env

An Environment object, to be used only when calling [“BOA::get_principal” on page 73](#) .

Return values

None.

CORBA module: ConstantDef Interface

| | |
|-----------------------------|---|
| Overview | The ConstantDef interface defines a named constant. |
| File name | somir.idl |
| Local-only | True |
| Ancestor interfaces | “Contained Interface” on page 79 |
| Exceptions | “CORBA::SystemException” on page |
| Supported operations | “ConstantDef::describe” on page 77 |
| | “IDLType::type” on page 127 |
| | “ConstantDef::type_def” on page 78 |

Intended Usage

The ConstantDef interface is used within the Interface Repository to represent a constant as defined within OMG IDL. An instance of a ConstantDef object defines the data type of the constant, the constant value, and the constant name. A ConstantDef object can be created using the create_constant operation of the Container interface.

IDL syntax

```

module CORBA
{
    interface ConstantDef:Contained
    {
        readonlyattribute TypeCode type;
        attribute IDLType type_def;
        attribute anyvalue;
    };
    struct ConstantDescription
    {
        Identifier name;
        RepositoryId id;
        VersionSpec version;
        TypeCode type;
        anyvalue;
    };
};

```

ConstantDef::describe

| | |
|---------------------------|---|
| Overview | The describe operation returns a structure containing information about a CORBA::ConstantDef Interface Repository object. |
| Original interface | "CORBA module: ConstantDef Interface" on page 76 |
| Exceptions | "CORBA::SystemException" on page |

Intended Usage

The inherited describe operation returns a structure (CORBA::Contained::Description) that contains information about a CORBA::ConstantDef Interface Repository object. The CORBA::Contained::Description structure has two fields: kind (CORBA::DefinitionKind data type), and value (CORBA::Any data type).

The kind of definition described by the returned structure is provided using the kind field, and the value field is a CORBA::Any that contains the description that is specific to the kind of object described. When the describe operation is invoked on a constant (CORBA::ConstantDef) object, the kind field is equal to CORBA::dk_Constant and the value field contains the CORBA::ConstantDescription structure.

IDL Syntax

```

struct ConstantDescription
{
    Identifier name;
    RepositoryId id;
    RepositoryId defined_in;
    VersionSpec version;
    TypeCode type;
    any value;
};
struct Description
{
    DefinitionKind kind;
    any value;
};
Description describe ();

```

Input parameters

None.

Return value

Description *

The returned value is a pointer to a CORBA::Contained::Description structure. The memory is owned by the caller and can be removed using delete.

Example

```
// C++
// assume that 'this_constant' has already been initialized
CORBA::ConstantDef * this_constant;
// retrieve a description of the constant
CORBA::ConstantDef::Description * returned_description;
returned_description = this_constant-> describe ();
// retrieve the constant description from the returned description
// structure
CORBA::ConstantDescription * constant_description;
constant_description = (CORBA::ConstantDescription *)
    returned_description-> value.value ();
```

ConstantDef::type_def

| | |
|---------------------------|---|
| Overview | The type_def operation returns a pointer to an IDLType that is representative of the type within a ConstantDef. |
| Original interface | "CORBA module: ConstantDef Interface" on page 76 |
| Exceptions | "CORBA::SystemException" on page |

Intended Usage

The type_def attribute within a ConstantDef references an IDLType that identifies the definition of the type of the constant. Both read and write type_def operations are supported, the parameters of which are identified below.

IDL Syntax

```
attribute IDLType type_def;
```

Read operations

Input parameters

none

Return values

CORBA::IDLType_ptr

The returned CORBA::IDLType * is a pointer to a copy of the information referenced by the type_def attribute. The object and the associated memory are owned by the caller and can be released by invoking CORBA::release.

Write operations

Input parameters

CORBA::IDLType_ptr

The CORBA::IDLType_ptr must reference a simple type¹.

Return values

none

Example

```
// C++
// assume that 'this_constant' and 'pk_long_def'
// have already been initialized
CORBA::ConstantDef * this_constant;
CORBA::PrimitiveDef * pk_long_def;
// set the type_def attribute of the constant
// to represent a CORBA::Long
this_constant-> type_def (pk_long_def);
// retrieve the type_def attribute from the constant
CORBA::IDLType * constants_type_def;
constants_type_def = this_constant-> type_def();
```

ConstantDef::value

| | |
|---------------------------|--|
| Overview | The value read and write operations allow the access and update of the value attribute of a ConstantDef. |
| Original interface | "CORBA module: ConstantDef Interface" on page 76 |

CORBA::pk_string

Intended Usage

The value attribute contains the value of the constant, not the computation of the value (for example, the fact that it was defined as "1+2"). Read and write value operations are provided with parameters as defined below.

IDL Syntax

```
any value;
```

Read operations

Input parameters

none

Return values

CORBA::Any *

The returned pointer to a CORBA::Any data type represents the value attribute of the constant. The object memory belongs to the caller, and can be removed by invoking delete.

Write operations

Input parameters

CORBA::Any & value

The value parameter is a reference to a CORBA::Any data type that provides the constant value to change the value attribute of the ConstantDef. When setting the value attribute, the TypeCode of the supplied CORBA::Any must be equal to the type attribute of the ConstantDef.

Return values

none

Example

```
// C++
// assume that 'constant_409' has already been initialized
CORBA::ConstantDef * constant_409;
// set the value '409' in the Any,
// and invoke the value operation to update the constant
CORBA::Any new_value;
new_value <<= (CORBA::Long) 409;
constant_409-> value (new_value);
// read the constant value from the ConstantDef
CORBA::Any * returned_value;
returned_value = constant_409-> value ();
```

CORBA module: Contained Interface

| | |
|---|--|
| Overview | The Contained interface is inherited by all Interface Repository interfaces that are contained by other objects. All objects within the Interface Repository, except the root object (Repository) and definitions of anonymous types (ArrayDef, StringDef, and SequenceDef), and primitive types are contained by other objects. |
| File name | somir.idl |
| Local-only | True |
| Ancestor interfaces | "IObject Interface" on page 138 |
| Exceptions | "CORBA::SystemException" on page |
| Supported operations | "Contained::absolute_name" on page 80 |
| "Contained::containing_repository" on page 81 | |

[“Contained::defined_in” on page 81](#)

[“Contained::describe” on page 82](#)

[“Contained::id” on page 83](#)

[“Contained::name” on page 84](#)

[“Contained::version” on page 84](#)

Intended Usage

The Contained interface is not itself instantiated as a means of accessing the Interface Repository. As an ancestor to certain Interface Repository objects, it provides a specific list of operations as noted below. Those Interface Repository objects that inherit (directly or indirectly) the operations defined in Contained include: ModuleDef, ConstantDef, StructDef, UnionDef, EnumDef, AliasDef, ExceptionDef, AttributeDef, OperationDef, and InterfaceDef.

IDL syntax

```
module CORBA
{
    typedef string VersionSpec;
    interface Contained:IRObject
    {
        // read/write interface
        attribute RepositoryId id;
        attribute Identifier name;
        attribute VersionSpec version;
        // read interface
        readonly attribute Container defined_in;
        readonly attribute ScopedName absolute_name;
        readonly attribute Repository containing_repository;
        struct Description
        {
            DefinitionKind kind;
            any value;
        };
        Description describe ();
    };
};
```

Contained::absolute_name

| | |
|---------------------------|---|
| Overview | The absolute_name operation retrieves the absolute ScopedName that identifies a Contained object within its enclosing Repository. |
| Original interface | “CORBA module: Contained Interface” on page 79 |
| Exceptions | “CORBA::SystemException” on page |

Intended Usage

The absolute_name attribute is an absolute ScopedName that identifies a Contained object uniquely within its enclosing Repository.

If the Container within which this object is defined is 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.

A read operation is provided to retrieve the absolute_name value for all Interface Repository objects that have a name attribute.

IDL Syntax

```
readonly attribute ScopedName absolute_name;
```

Input parameters

None

Return values

ScopedName

The returned value is a CORBA::ScopedName data type, the memory of which is owned by the caller. The caller can release this memory by invoking the CORBA::string_free function.

Example

```
// C++
// assume the interface_def_ptr has already been initialized
CORBA::InterfaceDef * interface_def_ptr;
// the following call returns the absolute name associated with the
// interface
CORBA::ScopedName returned_absolute_name;
returned_absolute_name = interface_def_ptr-> absolute_name();
```

Contained::containing_repository

| | |
|--------------------|--|
| Overview | The containing_repository attribute identifies the Repository that contains this object. |
| Original interface | "CORBA module: Contained Interface" on page 79 |
| Exceptions | "CORBA::SystemException" on page |

Intended Usage

A Contained object has a defined_in attribute that identifies the Container within which it is contained. The containing_repository attribute identifies the Repository that is eventually reached by recursively following the object's defined_in attribute.

The containing_repository attribute read operation retrieves a pointer to the Repository.

IDL Syntax

```
readonly attribute Repository containing_repository;
```

Input parameters

None

Return values

CORBA::Repository_ptr

A pointer to the Repository object is returned.

Example

```
// C++
// assume that 'interface_1' has already been initialized
CORBA::InterfaceDef * interface_1;
// retrieve a pointer to the Repository object
CORBA::Repository * repository_ptr;
repository_ptr = interface_1-> containing_repository();
```

Contained::defined_in

| | |
|--------------------|--|
| Overview | The defined_in operation returns the Container object of a Contained object. |
| Original interface | "CORBA module: Contained Interface" on page 79 |
| Exceptions | "CORBA::SystemException" on page |

Intended Usage

Contained objects 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 from which the object is inherited.

The `defined_in` operation is read-only and returns a pointer to a copy of the Container object identified by the `defined_in` attribute.

IDL Syntax

```
readonly attribute Container defined_in;
```

Input parameters

None

Return values

Container *

A pointer to the Container object of the `defined_in` attribute is returned. The caller owns the memory associated with this object, that can later be released using `CORBA::release`.

Example

```
// C++
// assume the interface_def_ptr has already been initialized
CORBA::InterfaceDef * interface_def_ptr;
// the following call returns the defined_in Container * for the interface
CORBA::Container * defined_in_container;
defined_in_container = interface_def_ptr-> defined_in ();
```

Contained::describe

| | |
|---------------------------|--|
| Overview | The describe operation returns a structure containing information about a <code>CORBA::Contained</code> Interface Repository object. |
| Original interface | "CORBA module: Contained Interface" on page 79 |
| Exceptions | "CORBA::SystemException" on page |

Intended Usage

The describe operation returns a structure that contains information about an Interface Repository object. The `CORBA::Description` structure has two fields: `kind` (`CORBA::Contained::DefinitionKind` data type), and `value` (`CORBA::Any` data type).

The kind of definition described by the returned structure is provided using the `kind` field, and the `value` field is a `CORBA::Any` that contains the description that is specific to the kind of object described. For example, if the describe operation is invoked on an attribute (`CORBA::AttributeDef`) object, the `kind` field is equal to `CORBA::dk_Attribute` and the `value` field contains the `AttributeDescription` structure.

The list of Interface Repository object types on which the describe operation may be called includes: modules (`CORBA::ModuleDefs`), constants (`CORBA::ConstantDefs`), type definitions (`CORBA::TypedDefDefs`), exceptions (`CORBA::ExceptionDefs`), attributes (`CORBA::AttributeDefs`), operations (`CORBA::OperationDefs`), and interfaces (`CORBA::InterfaceDefs`). For further information on the describe operation, please reference the describe operation descriptions for the object types listed above.

CORBA 2.1 specifies that the describe method on named IR objects will return a description structure that includes the repository ID of the container within which the IR object is defined. However, one common container has no repository ID, that is the Repository itself. In this situation, the IBM implementation returns a pointer to the empty string.

IDL Syntax

```
struct Description
{
    DefinitionKind kind;
    any value;
};
Description describe ();
```

Input parameters

None

Return values

Description *

The returned value is a pointer to a CORBA::Contained::Description structure. The memory is owned by the caller and can be removed using delete.

Example

```
// C++
// assume that 'this_attribute' has already been initialized
CORBA::AttributeDef * this_attribute;
// retrieve a description of the attribute
CORBA::AttributeDef::Description * returned_description;
returned_description = this_attribute-> describe ();
```

Contained::id

| | |
|--------------------|--|
| Overview | The id operations provide read and write capability for the id attribute of a Contained Interface Repository object. |
| Original interface | "CORBA module: Contained Interface" on page 79 |
| Exceptions | "CORBA::SystemException" on page |

Intended Usage

An object that is contained by another object has a unique id attribute that identifies it globally within the Interface Repository. The id read (Get) operation provides the ability to retrieve a copy of the id attribute, and the id write (Set) operation allows the unique id attribute to be changed.

IDL Syntax

```
void id (CORBA::RepositoryId repositoryid)
CORBA::RepositoryId id;
```

Read operations

Input parameters

none

Return values

CORBA::RepositoryId

The returned CORBA::RepositoryId is a copy of the id attribute of the Contained object. The associated memory is owned by the caller and can be freed by invoking CORBA::string_free.

Write operations

Input parameters

CORBA::RepositoryId new_id

The new_id parameter defines the new CORBA::RepositoryId value that will be used to uniquely identify the Contained object in the Interface Repository.

Return values

none

Example

```
// C++
// assume that 'this_union' has already been initialized
CORBA::UnionDef * this_union;
// change the 'id' attribute of the union (which is a contained object)
CORBA::RepositoryId new_rep_id = CORBA::string_dup ("new_rep_id_test");
this_union-> id (new_rep_id);
CORBA::string_free (new_rep_id);
// query the union to get a copy of the 'id' attribute
CORBA::RepositoryId returned_rep_id;
returned_rep_id = this_union-> id ();
```

Contained::name

| | |
|---------------------------|--|
| Overview | The name operations are used to read and write the name attribute of an Interface Repository object. |
| Original interface | "CORBA module: Contained Interface" on page 79 |
| Exceptions | "CORBA::SystemException" on page |

Intended Usage

An object that is contained by another object has a name attribute that identifies it uniquely within the enclosing Container object. Both Read and Write operations are supported, with parameters listed.

IDL Syntax

```
attribute identifier name;
```

Read operations

Input parameters

none

Return values

CORBA::Identifier

This operation returns a copy of the name of the object, that is owned by the caller. The caller may later free this memory by invoking CORBA::string_free.

Write operations

Input parameters

CORBA::Identifier name

A name that identifies the new name for the Interface Repository object.

Return values

none

Example

```
// C++
// assume 'interface_1' has already been created
CORBA::InterfaceDef * interface_1;
// establish a new name for the interface
interface_1-> name ("interface_409");
// retrieve the interface name
CORBA::Identifier retrieved_name;
retrieved_name = interface_1-> name ();
```

Contained::version

| | |
|---------------------------|---|
| Overview | The version read and write operations allow access and update of the version attribute of an Interface Repository Object. |
| Original interface | "CORBA module: Contained Interface" on page 79 |
| Exceptions | "CORBA::SystemException" on page |

Intended Usage

The version attribute distinguishes an object from the other versions. Both Read and Write methods are supported, with parameters listed below.

IDL Syntax

```
void version (CORBA::VersionSpec versionspec)
CORBA::VersionSpec version;
```

Read operations

Input parameters

none

Return values

CORBA::VersionSpec

The returned value is owned by the caller. It can be freed using CORBA::string_free.

Write operations

Input parameters

CORBA::VersionSpec version

The version parameter specifies the new version attribute value for the object.

Return values

none

Example

```
// C++
// assume that this interface has already been initialized.
CORBA::InterfaceDef * this_interface;
// change the version of this interface
this_interface-> version (<<2.0>>);
// retrieve the version from the interface
CORBA::VersionSpec returned_version;
returned_version = this_interface-> version();
```

CORBA module: Container Interface

| | |
|-----------------------------|--|
| Overview | The Container interface is used to form a containment hierarchy in the Interface Repository. |
| File name | somir.idl |
| Local-only | True |
| Ancestor interfaces | "IObject Interface" on page 138 |
| Exceptions | "CORBA::SystemException" on page 86 |
| Supported operations | "Container::contents" on page 86 |
| | "Container::create_alias" on page 87 |
| | "Container::create_constant" on page 88 |
| | "Container::create_enum" on page 89 |
| | "Container::create_exception" on page 90 |
| | "Container::create_interface" on page 91 |
| | "Container::create_module" on page 92 |
| | "Container::create_struct" on page 93 |
| | "Container::create_union" on page 94 |
| | "Container::describe_contents" on page 95 |
| | "Container::lookup" on page 96 |
| | "Container::lookup_name" on page 97 |

Intended Usage

A Container can contain any number of objects derived from the Contained interface. All Containers, except for Repository, are also derived from Contained. The Container interface is not itself instantiated as a means of accessing the Interface Repository. As an ancestor to certain Interface Repository objects, it provides a specific list of operations as noted below.

Those Interface Repository objects that inherit (directly or indirectly) the operations defined in Container include: Repository, ModuleDef, and InterfaceDef.

IDL syntax

```

module CORBA
{
    typedef sequence ContainedSeq;
    Interface Container:IRObject
    {
        //read interface
        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 DescriptionSeq;
        DescriptionSeq describe_contents (in DefinitionKind limit_type,
                                          in boolean exclude_inherited,
                                          in long max_returned_objs);

        //write interface
        ModuleDef create_module (in RepositoryId id,
                                in Identifier name,
                                in VersionSpec version);
        ConstantDef create_constant (in RepositoryId id,
                                    in Identifier name,
                                    in VersionSpec version,
                                    in IDLType type,
                                    in any value);
        StructDef create_struct (in RepositoryId id,
                                in Identifier name,
                                in VersionSpec version,
                                in StructMember Seqmembers);
        UnionDef create_union (in RepositoryId id,
                               in Identifier name,
                               in VersionSpec version,
                               in IDLType discriminator_type,
                               in UnionMemberSeq members);
        EnumDef create_enum (in RepositoryId id,
                             in Identifier name,
                             in VersionSpec version,
                             in EnumMemberSeq members);
        AliasDef create_alias (in RepositoryId id,
                               in Identifier name,
                               in VersionSpec version,
                               in IDLType original_type);
        InterfaceDef create_interface (in RepositoryId id,
                                       in Identifier name,
                                       in VersionSpec version,
                                       in InterfaceDefSeq base_interfaces);
    };
};

```

Container::contents

| | |
|---------------------------|--|
| Overview | The contents operation returns the list of objects directly contained by or inherited into the object. |
| Original interface | "CORBA module: Container Interface" on page 85 |
| Exceptions | "CORBA::SystemException" on page |

Intended Usage

The contents operation can be 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 within a specific module, and so on.

IDL Syntax

```

ContainedSeq contents (in DefinitionKind limit_type
                      in boolean exclude_inherited);

```

Input parameters

exclude_inherited

An object can be contained within another object because it is defined within the containing object (for example, an interface is contained within a module). It may also

be defined as contained because it is inherited by the containing object (for example, an operation may be contained by an interface because the interface inherits the operation from another interface).

When `exclude_inherited` is `TRUE`, inherited objects, if present, are not returned. If `exclude_inherited` is `FALSE`, all contained objects (whether contained due to inheritance or because they were defined within the object) are returned.

limit_type

If the `limit_type` is set to `CORBA::dk_all`, objects of all interface types are returned. For example, if this is an `InterfaceDef`, the attribute, operation, and exception objects are returned. If `limit_type` is set to a specific interface, only objects of that interface type are returned. For example, only attribute objects are returned if `limit_type` is set to `CORBA::dk_Attribute`. The accepted values for `limit_type` are: `CORBA::dk_all`, `CORBA::Attribute`, `CORBA::dk_Constant`, `CORBA::dk_Exception`, `CORBA::dk_Interface`, `CORBA::dk_Module`, `CORBA::dk_Operation`, `CORBA::dk_Typedef`, `CORBA::dk_Alias`, `CORBA::dk_Struct`, `CORBA::dk_Union`, and `CORBA::dk_Enum`.

Return values

ContainedSeq *

The returned value is a pointer to a `ContainedSeq` that is the list of `Contained` objects retrieved from the Interface Repository based upon the input criteria. The memory associated with the `ContainedSeq` is owned by the caller. The caller can invoke `delete` on the `ContainedSeq *` to delete the memory when no longer needed.

Example

```
// C++
// assume that 'repository_ptr' has already been initialized
CORBA::Repository * repository_ptr;
// retrieve all the objects contained by the Repository
CORBA::ContainedSeq * returned_sequence;
returned_sequence = repository_ptr-> contents (CORBA::dk_all, 0);
```

Container::create_alias

| | |
|---------------------------|---|
| Overview | The <code>create_alias</code> operation creates a new alias definition (<code>AliasDef</code>) in the Interface Repository. |
| Original interface | "CORBA module: Container Interface" on page 85 |
| Exceptions | "CORBA::SystemException" on page |

Intended Usage

The `create_alias` operation creates a new alias definition in the Interface Repository persistent database, and returns a pointer to a new `AliasDef` object associated with the alias definition. An `AliasDef` is typically used by the Interface Repository to represent an OMG IDL 'typedef'.

IDL Syntax

```
AliasDef create_alias (in Repositoryid id,
                     in Identifier name,
                     in VersionSpec version,
                     in IDLType original_type);
```

Input parameters

name

The name that will be associated with this `AliasDef` object in the Interface Repository.

original_type

The `original_type` identifies the original type to which this `AliasDef` refers. The `original_type` may be an instance of a `SequenceDef`, `ArrayDef`, `StringDef`, `PrimitiveDef`, `UnionDef`, `StructDef`, `AliasDef`, `EnumDef`, or `InterfaceDef`.

id

The id represents the CORBA::RepositoryId that will uniquely identify this AliasDef within the Interface Repository.

version

The version number that will be associated with this AliasDef object in the Interface Repository.

Return values**AliasDef_ptr**

A pointer to the created AliasDef object is returned to the caller. The memory associated with this object can later be released by invoking CORBA::release.

Example

```
// C++
// assume the 'repository_ptr' and 'structure_1' objects
// and these pointers have already been established
CORBA::Repository * repository_ptr;
CORBA::StructDef * structure_1;
// establish the id, name, and version values for the alias definition
CORBA::RepositoryId rep_id;
CORBA::Identifier name;
CORBA::VersionSpec version;
rep_id = CORBA::string_dup ("unique RepositoryID for this alias");
name = CORBA::string_dup ("alias_new");
version = CORBA::string_dup ("1.0");
// create the new alias for 'structure_1' . . .
CORBA::AliasDef * new_alias;
new_alias = repository_ptr-> create_alias (rep_id, name, version,
structure_1);
```

Container::create_constant

| | |
|---------------------------|---|
| Overview | The create_constant operation creates a new ConstantDef object. |
| Original interface | "CORBA module: Container Interface" on page 85 |
| Exceptions | "CORBA::SystemException" on page |

Intended Usage

The create_constant operation creates a new ConstantDef object with the specified type and value. A representation of the new ConstantDef object is created in the Interface Repository persistent database and a pointer to the memory representation of the ConstantDef object is returned to the caller.

IDL Syntax

```
ConstantDef create_constant (in RepositoryId Id,
                             in Identifier name,
                             in VersionSpec version,
                             in IDLType type,
                             in any value);
```

Input parameters**value**

The value parameter is an CORBA::Any reference. The Any contains the value of the constant.

name

The name that is associated with this ConstantDef object in the Interface Repository.

type

The type parameter is a CORBA::IDLType * that specifies the type of the ConstantDef. The type should be a CORBA::PrimitiveDef object of a simple type (pk_short, pk_long, pk_ushort, pk_ulong, pk_float, pk_double, pk_boolean, pk_char, pk_wchar, pk_string, pk_wstring, or pk_octet).

id

The id represents the CORBA::RepositoryId that will uniquely identify this ConstantDef

within the Interface Repository.

version

The version number that will be associated with this ConstantDef object in the Interface Repository.

Return values

ConstantDef_ptr

A pointer to the created ConstantDef object is returned to the caller. The memory associated with this object can later be released by invoking CORBA::release.

Example

```
// C++
// repository_ptr and module_one has already been initialized . . .
CORBA::Repository * repository_ptr;
CORBA::ModuleDef * module_one;
CORBA::RepositoryId constants_rep_id;
CORBA::Identifier constants_name;
CORBA::VersionSpec version;
CORBA::ConstantDef * constant_def_one;
CORBA::Any constants_value;
CORBA::PrimitiveDef * primitive_long;
// establish the id, name, and version values for the constant
constants_rep_id = CORBA::string_dup ("unique RepositoryID for my
constant");
constants_name = CORBA::string_dup ("constant_of_2001");
version = CORBA::string_dup ("1.0");
// establish the Any with a 'value' of 2001
constants_value <=<= (CORBA::Long) 2001;
// create a PrimitiveDef that represents a CORBA::Long data type
primitive_long = repository_ptr-> get_primitive (CORBA::pk_long);
// create the new constant that will be contained in module_one
constant_def_one = module_one-> create_constant (constants_rep_id,
constants_name, version, primitive_long, constants_value);
```

Container::create_enum

| | |
|---------------------------|---|
| Overview | The create_enum operation creates a new enumeration definition (EnumDef) in the Interface Repository. |
| Original interface | "CORBA module: Container Interface" on page 85 |
| Exceptions | "CORBA::SystemException" on page |

Intended Usage

The create_enum operation creates a new enumeration definition in the Interface Repository persistent database, and returns a pointer to a new EnumDef object associated with the enumeration definition.

IDL Syntax

```
EnumDef create_enum (in RepositoryId id,
                    in Identifier name,
                    in VersionSpec version,
                    in EnumMemberSeq members);
```

Input parameters

members

This is a reference to a CORBA::EnumMemberSeq that provides the list of the elements will comprise the new enumeration (EnumDef). The length of the CORBA::EnumMemberSeq must be greater than zero. The CORBA::EnumMemberSeq contains a distinct name for each possible value of the enumeration.

name

The name that will be associated with this EnumDef object in the Interface Repository.

id

The id represents the CORBA::RepositoryId that will uniquely identify this EnumDef within the Interface Repository.

version

The version number that will be associated with this EnumDef object in the Interface Repository.

Return values

EnumDef_ptr

A pointer to the created EnumDef object is returned to the caller. The memory associated with this object can later be released by invoking CORBA::release.

Example

```
// C++
// assume that 'repository_ptr' had already been initialized . . .
CORBA::Repository * repository_ptr;
// establish the id, name, and version values for the enumeration
CORBA::RepositoryId rep_id;
CORBA::Identifier name;
CORBA::VersionSpec version;
rep_id = CORBA::string_dup ("unique RepositoryID for my enumeration");
name = CORBA::string_dup ("enumeration new");
version = CORBA::string_dup ("1.0");
// instantiate an EnumMemberSeq and set the length to 2
CORBA::EnumMemberSeq enum_members;
enum_members.length(2);
// establish the EnumMemberSeq to represent an enumeration with two
// elements
enum_members[0] = (char *) CORBA::string_dup ("enum_value_0");
enum_members[1] = (char *) CORBA::string_dup ("enum_value_1");
// create the new enumeration . . .
CORBA::EnumDef * new_enum;
new_enum = repository_ptr-> create_enum (rep_id, name, version,
enum_members);
```

Container::create_exception

| | |
|--------------------|---|
| Overview | The create_exception operation returns a new exception definition (CORBA::ExceptionDef) contained in the CORBA::Container on which it is invoked. |
| Original interface | "CORBA module: Container Interface" on page 85 |
| Exceptions | "CORBA::SystemException" on page |

Intended Usage

The create_exception operation returns a new CORBA::ExceptionDef contained in the CORBA::Container on which it is invoked. A representation of the new CORBA::ExceptionDef object is created in the Interface Repository persistent database and a pointer to the memory representation of the CORBA::ExceptionDef object is returned to the caller.

The id, name, version, and members attributes are set as specified. The type attribute is also set.

An error is returned if an object with the specified id already exists within the Interface Repository, or if an object with the specified name already exists within the CORBA::Container on which the create_exception is invoked.

IDL Syntax

```
ExceptionDef * create_exception(RepositoryId id
                               Identifier name
                               VersionSpec version
                               StructMemberSeq & members);
```

Input parameters

name

The name that will be associated with this CORBA::ExceptionDef object in the Interface Repository.

id

The id represents the CORBA::RepositoryId that will uniquely identify this CORBA::ExceptionDef within the Container.

members

The members parameter defines the members of the exception definition. Each element of the members parameter has 3 fields. The name field is the name of the

element. The type field (a reference to a CORBA::TypeCode *) is not used for create_exception and should be set to CORBA::_tc_void. The type_def field references a CORBA::IDLType * that defines the type definition of the member element.

The members parameter can have a length of zero to indicate that the exception definition has no members.

version

The version number that will be associated with this CORBA::ExceptionDef object in the Interface Repository.

Return values

ExceptionDef *

The return value is a pointer to the newly created CORBA::ExceptionDef. The memory is owned by the caller and can be released using CORBA::release.

Example

```
// C++
// assume that 'this_module' and 'this_struct'
// have already been initialized
CORBA::ModuleDef * this_module;
CORBA::StructDef * this_struct;
// establish the 'create_exception' rep_id, name, and version parameters
CORBA::RepositoryId rep_id = "UniqueRepositoryId";
CORBA::Identifier name = "this_exception";
CORBA::VersionSpec version = "1.0";
// establish and initialize a CORBA::StructMemberSeq
// with which to create the CORBA::ExceptionDef
CORBA::StructMemberSeq members_list;
members_list length (1);
members_list[0].name = CORBA::string_dup ("exception_0");
members_list[0].type = CORBA::_tc_void;
members_list[0].type_def = this_struct;
// create the exception definition
this_module-> create_exception ( rep_id, name, version, members_list);
delete (members_list);
//cleanup.
```

Container::create_interface

| | |
|---------------------------|---|
| Overview | The create_interface operation is used to create a new interface definition (InterfaceDef) within the Interface Repository. |
| Original interface | "CORBA module: Container Interface" on page 85 |
| Exceptions | "CORBA::SystemException" on page |

Intended Usage

The create_interface operation returns a new empty InterfaceDef with the specified base_interfaces. Type, exception, and constant definitions can be added using the Container::create_, Container::create_exception, and Container::create_constant operations respectively on the new InterfaceDef. OperationDefs can be added using InterfaceDef::create_operation and AttributeDefs can be added using Interfacedef::create_attribute. Definitions can also be added using the Contained::move operation.

IDL Syntax

```
InterfaceDef create_interface (in RepositoryId id,
                              in Identifier name,
                              in VersionSpec version,
                              in InterfaceDefSeq base_interfaces);
```

Input parameters

name

The name that will be associated with this InterfaceDef object in the Interface Repository.

base_interface

The `base_interfaces` parameter lists all of the interfaces from which this interface inherits.

id

The `id` represents the `CORBA::RepositoryId` that will uniquely identify this `InterfaceDef` within the Interface Repository.

version

The version number that will be associated with this `InterfaceDef` object in the Interface Repository.

Return values

InterfaceDef_ptr

A pointer to the created `InterfaceDef` object is returned to the caller. The memory associated with this object can later be released by invoking `CORBA::release`.

Example

```
// C++
// assume that 'module_1', 'interface_A', and 'interface_B'
// have already been initialized
CORBA::ModuleDef * module_1;
CORBA::InterfaceDef * interface_A;
CORBA::InterfaceDef * interface_B;
// establish the id, name, and version values for the interface
CORBA::RepositoryId rep_id;
CORBA::Identifier name;
CORBA::VersionSpec version;
rep_id = CORBA::string_dup ("unique RepositoryID for my interface");
name = CORBA::string_dup ("interface_C");
version = CORBA::string_dup ("1.0");
// establish the base interfaces from which the new interface will
// inherit
CORBA::InterfaceDefSeq base_interfaces;
base_interfaces.length(2);
base_interfaces[0] = CORBA::InterfaceDef::_duplicate (interface_A);
base_interfaces[1] = CORBA::InterfaceDef::_duplicate (interface_B);
// create a new interface 'interface_C' . . .
CORBA::InterfaceDef * interface_C;
interface_C = module_1-> create_interface (rep_id, name, version,
base_interfaces);
```

Container::create_module

| | |
|---------------------------|--|
| Overview | The <code>create_module</code> operation creates a new module definition (<code>ModuleDef</code>) in the Interface Repository. |
| Original interface | "CORBA module: Container Interface" on page 85 |
| Exceptions | "CORBA::SystemException" on page |

Intended Usage

The `create_module` operation returns a new empty `ModuleDef` object after it creates a corresponding new module representation in the Interface Repository persistent database. Definitions can be added using `Container::create_` operations on the new module, or by using the `Contained::move` operation.

IDL Syntax

```
ModuleDef create_module (in RepositoryId id,
in Identifier name,
in VersionSpec version);
```

Input parameters

name

The name that will be associated with this `ModuleDef` object in the Interface Repository.

id

The `id` represents the `CORBA::RepositoryId` that will uniquely identify this `ModuleDef` within the Interface Repository.

version

The version number that will be associated with this `ModuleDef` object in the Interface

Repository.

Return values ModuleDef_ptr

A pointer to the created ModuleDef object is returned to the caller. The memory associated with this object can later be released by invoking CORBA::release.

Example

```
// C++
// repository_ptr has already been initialized . . .
CORBA::Repository * repository_ptr;
CORBA::ModuleDef * new_module;
CORBA::RepositoryId modules_rep_id;
CORBA::Identifier modules_name;
CORBA::VersionSpec version;
// establish the id, name, and version values for the module
modules_rep_id = CORBA::string_dup ("unique RepositoryID for my module");
modules_name = CORBA::string_dup ("module new");
version = CORBA::string_dup ("1.0");
// create the new module
new_module = repository_ptr-> create_module (modules_rep_id, modules_name
```

Container::create_struct

| | |
|--------------------|---|
| Overview | The create_struct operation creates a new structure definition (StructDef) in the Interface Repository. |
| Original interface | "CORBA module: Container Interface" on page 85 |
| Exceptions | "CORBA::SystemException" on page |

Intended Usage

The create_struct operation creates a new structure definition in the Interface Repository persistent database, and returns a pointer to a new StructDef object associated with the struct definition.

IDL Syntax

```
StructDef create_struct (in RepositoryId id,
                        in Identifier name,
                        in VersionSpec version,
                        in StructMemberSeq members);
```

Input parameters

name

The name that will be associated with this StructDef object in the Interface Repository.

id

The id represents the CORBA::RepositoryId that will uniquely identify this StructDef within the Interface Repository.

members

This is a reference to a CORBA::StructMemberSeq that provides the list of the elements will comprise the new structure (StructDef). The length of the CORBA::StructMemberSeq must be greater than zero.

Each CORBA::StructMember within the CORBA::StructMemberSeq has 3 fields. The name field identifies the name of the StructMember. The type field of the StructMember is ignored by create_struct, and should be set to CORBA::_tc_void. The type_def field is a CORBA::IDLType * that represents the type definition of the StructMember.

version

The version number that will be associated with this StructDef object in the Interface Repository.

Return values StructDef_ptr

A pointer to the created StructDef object is returned to the caller. The memory

associated with this object can later be released by invoking CORBA::release.

Example

```
// C++
// assume 'primitive_long', 'primitive_double',
// and 'repository_ptr' have already been instantiated
CORBA::PrimitiveDef * primitive_double;
CORBA::PrimitiveDef * primitive_long;
CORBA::Repository * repository_ptr;
// establish the id, name, and version values for the structure
CORBA::RepositoryId rep_id;
CORBA::Identifier name;
CORBA::VersionSpec version;
rep_id = CORBA::string_dup ("unique RepositoryId for my structure");
name = CORBA::string_dup ("structure_new");
version = CORBA::string_dup ("1.0");
/// instantiate a StructMemberSeq and set the length to 2
CORBA::StructMemberSeq struct_members;
struct_members.length(2);
// establish the StructMemberSeq to represent a structure with two
// elements: a CORBA::Double called 'x', and a CORBA::Long called 'y'.
struct_members[0].name = CORBA::string_dup ("x");
struct_members[0].type_def =
    CORBA::IDLType::_duplicate (primitive_double);
struct_members[1].name = CORBA::string_dup ("y");
struct_members[1].type_def =
    CORBA::IDLType::_duplicate (primitive_long);
// create the new structure . . .
CORBA::StructDef * new_struct;
new_struct = repository_ptr-> create_struct (rep_id, name, version,
    struct_members);
```

Container::create_union

| | |
|---------------------------|---|
| Overview | The create_union operation creates a new union definition (UnionDef) in the Interface Repository. |
| Original interface | "CORBA module: Container Interface" on page 85 |
| Exceptions | "CORBA::SystemException" on page |

Intended Usage

The create_union operation creates a new union definition in the Interface Repository persistent database, and returns a pointer to a new UnionDef object associated with the union definition.

IDL Syntax

```
UnionDef create_union (in RepositoryId id,
                      in Identifier name,
                      in VersionSpec version,
                      in IDLType discriminator_type,
                      in UnionMemberSeq members);
```

Input parameters

name

The name that will be associated with this UnionDef object in the Interface Repository.

discriminator_type

This is a CORBA::IDLType * that identifies the union's discriminator type. The discriminator type can be a PrimitiveDef (with a primitive kind of CORBA::pk_long, CORBA::pk_short, CORBA::pk_ulong, CORBA::pk_ushort, CORBA::pk_char, CORBA::pk_wchar, or CORBA::pk_boolean) or an EnumDef (which represents an enumerator definition).

id

The id represents the CORBA::RepositoryId that will uniquely identify this UnionDef within the Interface Repository.

members

This is a reference to a CORBA::UnionMemberSeq that provides the list of the elements that will comprise the new union (UnionDef). The length of the CORBA::UnionMemberSeq must be greater than zero.

Each CORBA::UnionMember within the CORBA::UnionMemberSeq has 4 fields. The

name field identifies the name of the UnionMember. The type field of the UnionMember is ignored by create_union, and should be set to CORBA::_tc_void. The label field of each UnionMember is a CORBA::Any data type that represents a distinct value of the discriminator_type (a label of type CORBA::Octet and value 0 indicates the default union member). The type_def field is a CORBA::IDLType * that represents the type definition of the UnionMember.

version

The version number that will be associated with this UnionDef object in the Interface Repository.

Return values

UnionDef_ptr

A pointer to the created UnionDef object is returned to the caller. The memory associated with this object can later be released by invoking CORBA::release.

Example

```
// C++
// assume 'primitive_long', 'primitive_double'
// 'structure_1' and 'repository_ptr' have already been instantiated
CORBA::PrimitiveDef * primitive_long;
CORBA::PrimitiveDef * primitive_double;
CORBA::StructDef * structure_1;
CORBA::Repository * repository_ptr;
// establish the id, name, and version values for the union
CORBA::RepositoryId rep_id;
CORBA::Identifier name;
CORBA::VersionSpec version;
rep_id = CORBA::string_dup ("unique RepositoryId for my union");
name = CORBA::string_dup ("union new");
version = CORBA::string_dup ("1.0");
// the discriminator type is in this case CORBA::Long
CORBA::IDLType * discriminator_type;
discriminator_type = primitive_long;
// instantiate a UnionMemberSeq and set the length to 2
CORBA::UnionMemberSeq union_members;
union_members.length(2);
// establish the UnionMemberSeq to represent a union with two
// elements: a CORBA::Double called 'x', and a previously created
// structure called 'y'.
union_members[0].name = CORBA::string_dup ("x");
union_members[0].type_def =
    CORBA::IDLType::_duplicate (primitive_double);
union_members[0].label <= (CORBA::Long) 1;
union_members[1].name = CORBA::string_dup ("y");
union_members[1].type_def = CORBA::IDLType::_duplicate (structure_1);
union_members[1].label <= (CORBA::Long) 2;
// create the new union . . .
CORBA::UnionDef * new_union;
new_union = repository_ptr-> create_union (rep_id, name, version,
    discriminator_type, union_members);
```

Container::describe_contents

| | |
|---------------------------|---|
| Overview | The describe_contents operation combines the contents operation and the describe operation. |
| Original interface | "CORBA module: Container Interface" on page 85 |
| Exceptions | "CORBA::SystemException" on page |

Intended Usage

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).

IDL Syntax

```
struct Description
{
    Contained contained_object;
    DefinitionKind kind;
    any value;
};
DescriptionSeq describe_contents (in DefinitionKind limit_type,
    in boolean exclude_inherited,
    in long max_returned_objs);
```

Input parameters

exclude_inherited

CORBA::Contained objects have a defined_in attribute that identifies the CORBA::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 the exclude_inherited parameter is set to TRUE, inherited objects (if there are any) are not returned. If exclude_inherited is set to FALSE, all contained objects are returned (whether contained due to inheritance or because they were defined within the object).

limit_type

The limit_type identifies the interface types for which descriptions are returned. If limit_type is set to CORBA::dk_all, objects of all interface types within the CORBA::Container are returned. If limit_type is set to a specific interface, only objects of that interface are returned.

Valid values for limit_type include CORBA::dk_all, CORBA::dk_Attribute, CORBA::dk_Constant, CORBA::dk_Exception, CORBA::dk_Interface, CORBA::dk_Module, CORBA::dk_Operation, CORBA::dk_Typedef, CORBA::dk_Union, CORBA::dk_Alias, CORBA::dk_Struct, and CORBA::dk_Enum.

max_returned_objs

The max_returned_objs parameter 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 that all contained objects should be returned.

Return values

DescriptionSeq *

The returned value is a pointer to a sequence of descriptions of Interface Repository objects. The memory is owned by the caller and can be removed using delete.

Example

```
// C++
// assume that 'this_module' has already been initialized
CORBA::ModuleDef * this_module;
// retrieve information about all objects contained
// within the module using 'describe_contents'
CORBA::Container::DescriptionSeq * returned_descriptions;
returned_descriptions =
this_module-> describe_contents (CORBA::dk_all, 1, -1);
```

Container::lookup

| | |
|---------------------------|--|
| Overview | The lookup operation locates a definition relative to this container given a scoped name using OMG IDL's name scoping rules. |
| Original interface | "CORBA module: Container Interface" on page 85 |
| Exceptions | "CORBA::SystemException" on page |

Intended Usage

The lookup operation locates a definition relative to this container given a scoped name using OMG IDL's name scoping rules. An absolute scoped name (beginning with "::") locates the definition relative to the enclosing Repository. If no object is found, a nil object reference is returned.

IDL Syntax

```
Contained lookup (in ScopedName search_name);
```

Input parameters

search_name

The search_name is the scoped name of the object using OMG IDL's name scoping rules. This name is used as the search criteria for locating the object within the Interface Repository.

Return values

Contained_ptr

The return value is a pointer to a CORBA::Contained object resulting from the search. If the search_name was not located within the Interface Repository, a nil object is returned. If a non nil CORBA::Contained object pointer is returned, the memory associated with the object is owned by the caller and can be released by invoking CORBA::release.

Example

```
// C++
// assume that 'module_1' has already been initialized
CORBA::ModuleDef * module_1;
// use the scoped name to lookup an operation . . .
CORBA::Contained * ret_contained;
ret_contained = module_1-> lookup ( "Module2::Interface6::Operation7" );
```

Container::lookup_name

| | |
|--------------------|--|
| Overview | The lookup_name operation is used to locate an object by name within a particular object or within the objects contained by that object. |
| Original interface | "CORBA module: Container Interface" on page 85 |
| Exceptions | "CORBA::SystemException" on page |

Intended Usage

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 parameters to the lookup_name operation specify the name for the search, the number of levels to search, the type of objects to be examined in the search, and whether containment by inheritance should be included.

IDL Syntax

```
ContainedSeq lookup_name (in Identifier search_name,
                        in long levels_to_search,
                        in DefinitionKind limit_type,
                        in boolean exclude_inherited);
```

Input parameters

exclude_inherited

An object can be contained within another object because it is defined within the containing object (for example, an interface is contained within a module). It may also be defined as contained because it is inherited by the containing object (for example, an operation may be contained by an interface because the interface inherits the operation from another interface).

When exclude_inherited is TRUE, inherited objects, if present, are not returned. If exclude_inherited is FALSE, all contained objects (whether contained due to inheritance or because they were defined within the object) are returned.

limit_type

The limit_type parameter indicates which type of object should be examined while performing the name search. The accepted values for limit_type are: CORBA::dk_all, CORBA::dk_Attribute, CORBA::dk_Constant, CORBA::dk_Exception, CORBA::dk_Interface, CORBA::dk_Module, CORBA::dk_Operation, CORBA::dk_Typedef, CORBA::dk_Alias, CORBA::dk_Struct, CORBA::dk_Union, and CORBA::dk_Enum. If the limit_type is CORBA::dk_all, objects of all interface types are

returned (for example, attributes, operations, and exceptions are all returned). If `limit_type` is set to a specific interface, only objects of that interface are returned.

search_name

The `search_name` specifies the name for which the search is to be made.

levels_to_search

The `levels_to_search` parameter controls the the depth of the search. When `levels_to_search` is set to -1 the current object is searched as well as all contained objects. Setting `levels_to_search` to 1 will limit the search to the current object only.

Return values

ContainedSeq *

The returned value is a pointer to a `ContainedSeq` that is the list of `Contained` objects retrieved from the Interface Repository based upon the input criteria. The memory associated with the `ContainedSeq` is owned by the caller. The caller can invoke `delete` on the `ContainedSeq *` to delete the memory when no longer needed.

Example

```
// C++
// assume 'repository_ptr' is already initialized
CORBA::Repository * repository_ptr;
// search for a specific interface name
CORBA::ContainedSeq * cont_seq;
cont_seq = repository_ptr->lookup_name ("Interface1", -1,
CORBA::dk_Interface, 0);
```

CORBA module: Context Class

| | |
|--------------------------|--|
| Overview | Contains a list of properties that represent information about the client environment. |
| File name | context.h |
| Supported methods | "Context::_duplicate" on page 99 "Context::_nil" on page 99 "Context::context_name" on page 99 "Context::create_child" on page 100 "Context::delete_values" on page 100 "Context::get_values" on page 101 "Context::parent" on page 101 "Context::set_one_value" on page 102 "Context::set_values" on page 102 |

Intended Usage

The `Context` class is used to pass information from the client environment to the server environment, specifically information that is inconvenient to pass as method parameters. The information is specified as a list of properties. Each property consists of a name and a string value associated with that name. An IDL operation specification may contain a clause specifying context properties that should be passed to the server when the method is invoked. The `Context` object associated with an operation is passed as a distinguished parameter. The `ORB::get_default_context` method is called to get a reference to the default process context. The `Context` class provides methods to add and delete properties, as well as query information about a context.

Contexts may be "chained" together to achieve a particular default behavior. Property

searches on a child context recursively look in the parent context. Contexts may optionally be named. A context name can be used to specify a starting search scope.

Context::_duplicate

| | |
|----------------|---|
| Overview | Duplicates a Context object. |
| Original class | "CORBA::Context" on page 98 |

Intended Usage

This method is intended to be used by client and server applications to duplicate a reference to a Context object. Both the original and the duplicate reference should subsequently be released using `CORBA::release(Context_ptr)`.

IDL Syntax

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

Input parameters

p

The Context object to be duplicated. The reference can be nil, in which case the return value will also be nil.

Return values

CORBA::Context_ptr

The new Context object reference. This value should subsequently be released using `CORBA::release(Context_ptr)`.

Context::_nil

| | |
|----------------|---|
| Overview | Returns a nil CORBA::Context reference. |
| Original class | "CORBA::Context" on page 98 |

Intended Usage

This method is intended to be used by client and server applications to create a nil Context reference.

IDL Syntax

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

Input parameters

None

Return values

CORBA::Context_ptr

A nil Context reference.

Context::context_name

| | |
|----------------|---|
| Overview | Retrieves the name of a context. |
| Original class | "CORBA::Context" on page 98 |

Intended Usage

Context objects may optionally be named. A context name can be used to specify a starting search scope. The `context_name` method retrieves the name of a context.

IDL Syntax

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

Input parameters

None

Return values

const char *

A pointer to the name of the context, if any, or a null pointer. Ownership of the return value is maintained by the Context; the return value must not be freed by the caller.

Context::create_child

| | |
|----------------|--|
| Overview | Creates a child Context object. |
| Original class | "CORBA::Context" on page 98 |
| Exceptions | "CORBA::SystemException" on page |

Intended Usage

Context objects may be "chained" together to achieve a particular default behavior. Property searches on a child context recursively look in the parent context. The create_child method creates a new child Context object. The child context is chained to the parent context, which is the target object.

IDL Syntax

```
CORBA::Status create_child(const char *ctx_name,  
                           CORBA::Context_ptr &child_ctx);
```

Input parameters

ctx_name

The name of the child context to be created, if any, or a null pointer. If specified, the input context name should follow the rules for OMG IDL identifiers. The caller retains ownership of the input name (the Context makes its own copy).

child_ctx

A pointer for a Context object, passed by reference, to be updated by the create_child method to point to the newly created child context. Ownership of this parameter transfers to the caller and must be freed by calling CORBA::release(Context_ptr).

Return values

CORBA::Status

A zero return code indicates the child Context object was successfully created.

Context::delete_values

| | |
|----------------|--|
| Overview | Deletes one or more properties. |
| Original class | "CORBA::Context" on page 98 |
| Exceptions | "CORBA::SystemException" on page |

Intended Usage

The delete_values method deletes the specified properties from a Context object. If the property name has a trailing wildcard character ("*"), then all property names that match are deleted. Search scope is always limited to the specified context.

IDL Syntax

```
CORBA::Status delete_values(const char *prop_name);
```

Input parameters

prop_name

An identifier specifying the properties to be deleted. To specify multiple properties, pass an identifier with a trailing wildcard character. If a null pointer is passed for this parameter, a system exception is raised.

Return values

CORBA::Status

A zero return code indicates the properties were successfully deleted. If the input property name is not found, a system exception is raised.

Context::get_values

| | |
|----------------|--|
| Overview | Retrieves one or more property values. |
| Original class | "CORBA::Context" on page 98 |
| Exceptions | "CORBA::SystemException" on page |

Intended Usage

The `get_values` method retrieves the specified context property values. If the property name has a trailing wildcard character ("`*`"), then all matching properties and their values are returned. If a matching property is not found at the starting scope, the search optionally continues up the context tree until a match is found or all contexts in the chain have been exhausted.

IDL Syntax

```
CORBA::Status get_value(const char *start_scope,
                       CORBA::Flags op_flags,
                       const char *prop_name,
                       CORBA::NVList_ptr &values);
```

Input parameters

start_scope

An identifier specifying the name of the context at which the search should begin. If the search scope is not specified, the search begins with the target Context object. If the specified search scope is not found, a system exception is raised.

op_flags

`CORBA::CTX_RESTRICT_SCOPE` may be used to indicate searching is limited to the specified search scope. By default, the entire context tree is searched.

prop_name

An identifier specifying the name of the properties to return. To specify multiple properties, pass an identifier with a trailing wildcard character.

values

The pointer for an NVList object, passed by reference, to be updated by the `get_values` method to point to the resulting NVList. Ownership of the returned object transfers to the caller. Memory should be freed by calling `CORBA::release(NVList_ptr)`. The pointer for an NVList object, passed by reference, to be updated by the `get_values` method to point to the resulting NVList. Ownership of the returned object transfers to the caller. Memory should be freed by calling `CORBA::release(NVList_ptr)`.

Return values

CORBA::Status

A zero return code indicates that one or more property values were successfully returned. If no matching property name is found, -1 is returned.

Context::parent

| | |
|----------------|---|
| Overview | Retrieves the parent context. |
| Original class | "CORBA::Context" on page 98 |

Intended Usage

Context objects may be "chained" together to achieve a particular defaulting behavior.

Property searches on a child ontext recursively look in the parent context. The parent method retrieves the parent of the target Context object.

IDL Syntax

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

Input parameters

None

Return values

Context_ptr

A pointer to the parent context, if any, or a null pointer. Ownership of the return value is maintained by the Context; the return value must not be freed by the caller.

Context::set_one_value

| | |
|----------------|--|
| Overview | Adds a single property. |
| Original class | "CORBA::Context" on page 98 |
| Exceptions | "CORBA::SystemException" on page |

Intended Usage

The set_one_value method adds a single property to a context. If the input property name is not found in the property list, a new NamedValue (with the input property name and value) is added. If the input property name is found, the associated NamedValue is removed, then a new NamedValue (with the input property name and value) is added.

IDL Syntax

```
CORBA::Status set_one_value(const char *prop_name,  
                           const CORBA::Any &value);
```

Input parameters

prop_name

The name of the property to be added. Context properties follow the rules for OMG IDL identifiers. Property names should not end with an asterisk. If a null pointer is passed for this parameter, a system exception is raised. The caller retains ownership of the input string (the Context makes its own copy).

value

The address of the value of the property to be added. Currently, only strings are supported as property values. It is legal to pass a null pointer. The caller retains ownership of the input Any.

Return values

CORBA::Status

A zero return code indicates the property was successfully added.

Context::set_values

| | |
|----------------|--|
| Overview | Adds one or more properties. |
| Original class | "CORBA::Context" on page 98 |
| Exceptions | "CORBA::SystemException" on page |

Intended Usage

The set_values method adds one or more properties to a context. If an input property names is not found in the property list, a new NamedValue (wth the input property name and value) is added. If the input property name is found, the associated NamedValue is removed, then a new NamedValue (wth the input property name and value) is added.

IDL Syntax

```
CORBA::Status set_values(CORBA::NVList_ptr values);
```

Input parameters values

A pointer to an NVList containing the properties to be set. Context properties follow the rules for OMG IDL identifiers. The property names should not end with an asterisk. Property names must be non-null, or a system exception is raised. Currently, only strings are supported as property values. It is legal to pass a null property value. In the NVList, the flags field must be set to zero. The caller retains ownership of this parameter.

Return values CORBA::Status

A zero return code indicates the properties were successfully added.

CORBA module: ContextList Class

| | |
|--------------------------|---|
| Overview | Specifies the list of properties sent with a DII request. |
| File name | contextl.h |
| Supported methods | "ContextList::_duplicate" on page 103 |
| | "ContextList::_nil" on page 104 |
| | "ContextList::add" on page 104 |
| | "ContextList::add_consume" on page 104 |
| | "ContextList::count" on page 105 |
| | "ContextList::item" on page 105 |
| | "ContextList::remove" on page 106 |

Intended Usage

When a client assembles a Dynamic Invocation Interface request, a ContextList is optionally included. A ContextList specifies the list of properties sent with a request and is used to improve performance. When invoking a request without a ContextList, the ORB looks up context information in the Interface Repository. The ORB::create_context_list method is called to create an empty context list. The ContextList class provides methods to add and delete a property, as well as query information about a context list. Note that a context list contains only property names, not property values. Associations between property names and property values are maintained in a Context object. For additional information, see the Context and Request class descriptions.

ContextList::_duplicate

| | |
|-----------------------|--|
| Overview | Duplicates a ContextList object. |
| Original class | "CORBA::ContextList" on page 103 |

Intended Usage

This method is intended to be used by client and server applications to duplicate a reference to a ContextList object. Both the original and the duplicate reference should subsequently be released using CORBA::release(ContextList_ptr).

IDL Syntax

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

Input parameters p

The ContextList object to be duplicated. The reference can be nil, in which case the return value will also be nil.

Return values

CORBA::ContextList_ptr

The new ContextList object reference. This value should subsequently be released using CORBA::release(ContextList_ptr).

ContextList::_nil

| | |
|----------------|--|
| Overview | Returns a nil CORBA::ContextList reference. |
| Original class | "CORBA::ContextList" on page 103 |

Intended Usage

This method is intended to be used by client and server applications to create a nil ContextList reference.

IDL Syntax

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

Input parameters

None

Return values

CORBA::ContextList_ptr

A nil ContextList reference.

ContextList::add

| | |
|----------------|--|
| Overview | Adds a single property name to a context list. |
| Original class | "CORBA::ContextList" on page 103 |
| Exceptions | "CORBA::SystemException" on page |

Intended Usage

The add method is used by a client program to populate the ContextList associated with a DII request. The add method adds a single property name to a context list. The add and add_consume methods perform the same task but differ in memory management. The add method does not assume ownership of the input property name; the add_consume method does.

IDL Syntax

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

Input parameters

ctxt

The name of the property to be added. Property names follow the rules for OMG IDL identifiers and should not end with an asterisk. A system exception is raised if the input property name is null.

Return values

None

ContextList::add_consume

| | |
|----------------|--|
| Overview | Adds a single property name to a context list. |
| Original class | "CORBA::ContextList" on page 103 |

| | |
|------------|--|
| Exceptions | "CORBA::SystemException" on page |
|------------|--|

Intended Usage

The `add_consume` method is used by a client program to populate the `ContextList` associated with a DII request. The `add_consume` method adds a single property name to a context list. The `add` and `add_consume` methods perform the same task but differ in memory management. The `add_consume` method assumes ownership of the input property name; the `add` method does not. The caller must not access the memory referred to by the input parameter after it has been passed in.

IDL Syntax

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

Input parameters

ctxt

The name of the property to be added. Property names follow the rules for OMG IDL identifiers and should not end with an asterisk. The property name must be allocated using the `CORBA::string_alloc` method. Ownership of this parameter transfers to the `ContextList`. A system exception is raised if the input property name is null.

Return values

None

ContextList::count

| | |
|----------------|---|
| Overview | Retrieves the number of elements in a context list. |
| Original class | "CORBA::ContextList" on page 103 |

Intended Usage

The `count` method is used by a client program when querying the `ContextList` associated with a DII request. The `count` method returns the number of elements in a context list.

IDL Syntax

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

Input parameters

None

Return values

CORBA::ULong

The number of elements in the context list.

ContextList::item

| | |
|----------------|---|
| Overview | Retrieves the property name associated with an input index. |
| Original class | "CORBA::ContextList" on page 103 |
| Exceptions | "CORBA::SystemException" on page |

Intended Usage

The `item` method is used by a client program when querying the `ContextList` associated with a DII request. The `item` method returns the property name associated with an input index.

IDL Syntax

```
const char * item(CORBA::ULong index)
```

Input parameters

index

The index of the desired property name, starting at zero. A system exception is raised if the input index is greater than or equal to the number of elements in the context list.

Return values
const char *

The property name associated with the input index. Ownership of the return value is maintained by the ContextList; the return value must not be freed by the caller.

ContextList::remove

| | |
|-----------------------|---|
| Overview | Deletes the property name associated with an input index. |
| Original class | "CORBA::ContextList" on page 103 |
| Exceptions | "CORBA::SystemException" on page |

Intended Usage

The remove method is used by a client program when deleting an element of the context list associated with a DII request. The remove method deletes the property name associated with an input index. The remaining property names are re-indexed.

IDL Syntax

```
CORBA::Status remove(CORBA::ULong index);
```

Input parameters

index

The index of the property name to be deleted, starting at zero. A system exception is raised if the input index is greater than or equal to the number of elements in the context list.

Return values

CORBA::Status

A zero return code indicates the property name was successfully deleted.

CORBA module: CORBA Class

| | |
|--------------------------------------|--|
| Overview | Encompasses the interfaces and classes that make up the CORBA-compliant ORB, the TypeCode library, and the Interface Repository Framework. |
| File name | corba.h |
| Nested interfaces and classes | "AliasDef Interface" on page 54 "Any Class" on page 55 "ArrayDef Interface" on page 61 "AttributeDef Interface" on page 64 "BOA Class" on page 67 "ConstantDef Interface" on page 76 "Contained Interface" on page 79 "Container Interface" on page 85 "Context Class" on page 98 "ContextList Class" on page 103 "Current Class" on page 115 "BOA::DynamicImplementation Class" on page 75 |

| | |
|---|---------------------------|
| "EnumDef Interface" on page 116 | |
| "Environment Class" on page 118 | |
| "Exception Class" on page 120 | |
| "ExceptionDef Interface" on page 121 | |
| "ExceptionList Class" on page 123 | |
| "IDLType Interface" on page 127 | |
| "ImplementationDef Interface" on page 128 | |
| "ImplRepository Class" on page 130 | |
| "InterfaceDef Interface" on page 132 | |
| "IObject Interface" on page 138 | |
| "ModuleDef Interface" on page 140 | |
| "NamedValue Class" on page 141 | |
| "NVList Class" on page 144 | |
| "Object Class" on page 150 | |
| "OperationDef Interface" on page 160 | |
| "ORB Class" on page 167 | |
| "Policy Interface" on page 189 | |
| "PrimitiveDef Interface" on page 190 | |
| "Principal Interface" on page 191 | |
| "Repository Interface" on page 191 | |
| "Request Class" on page 195 | |
| "RequestSeq Class" on page 206 | |
| "SequenceDef Interface" on page 208 | |
| "ServerRequest Class" on page 211 | |
| "StringDef Interface" on page 215 | |
| "StructDef Interface" on page 216 | |
| "SystemException Class" on page 218 | |
| "TypeCode Class" on page 221 | |
| "TypedefDef Interface" on page 228 | |
| "UnionDef Interface" on page 229 | |
| "UnknownUserException Class" on page 232 | |
| "UserException Class" on page 234 | |
| "WstringDef Interface" on page 235 | |
| Supported methods | "CORBA::_boa" on page 109 |
| "CORBA::is_nil" on page 109 | |
| "CORBA::ORB_init" on page 110 | |
| "CORBA::release" on page 111 | |
| "CORBA::string_alloc" on page 112 | |
| "CORBA::string_dup" on page 113 | |
| "CORBA::string_free" on page 113 | |

["CORBA::wstring_alloc" on page 114](#)

["CORBA::wstring_dup" on page 114](#)

["CORBA::wstring_free" on page 115](#)

Intended Usage

The intended use of the CORBA class is the same as that of the CORBA module; see ["CORBA module in Object Request Broker" on page 52](#) .

Types

```
typedef unsigned char Boolean;
typedef unsigned char Char;
typedef wchar_t WChar;
typedef unsigned char Octet;
typedef short Short;
typedef unsigned short UShort;
typedef long Long;
typedef unsigned long ULong;
typedef float Float;
typedef double Double;
typedef char* String;
typedef WChar_t* WString;
typedef void Void;
typedef ULong Status;
typedef ULong Flags;
typedef Environment* Environment_ptr;
typedef Request* Request_ptr;
typedef ServerRequest* ServerRequest_ptr;
typedef Object* Object_ptr;
typedef BOA* BOA_ptr;
typedef ORB* ORB_ptr;
typedef Context* Context_ptr;
typedef ContextList* ContextList_ptr;
typedef Exception* Exception_ptr;
typedef ExceptionList* ExceptionList_ptr;
typedef NamedValue* NamedValue_ptr;
typedef NVList* NVList_ptr;
typedef ImplementationDef* ImplementationDef_ptr;
typedef ImplRepository* ImplRepository_ptr;
typedef InterfaceDef* InterfaceDef_ptr;
typedef OperationDef* OperationDef_ptr;
typedef Principal* Principal_ptr;
typedef TypeCode* TypeCode_ptr;
typedef Any* Any_ptr;
typedef char* ORBid;
typedef UShort ServiceType;
typedef ULong ServiceOption;
typedef ULong ServiceDetailType;
struct ServiceDetail {
    ServiceDetailType service_detail_type;
    _IDL_SEQUENCE_Octet service_detail;
    void encodeOp (Request &r) const;
    void decodeOp (Request &r);
    void decodeInOutOp (Request &r);
}; // struct ServiceDetail
struct ServiceInformation {
    _IDL_SEQUENCE_CORBA_ULong_0 service_options;
    SeqServiceDetail service_details;
    void encodeOp (Request &r) const;
    void decodeOp (Request &r);
    void decodeInOutOp (Request &r);
}; // struct ServiceInformation
enum CompletionStatus { COMPLETED_YES, COMPLETED_NO, COMPLETED_MAYBE };
enum exception_type { NO_EXCEPTION, USER_EXCEPTION, SYSTEM_EXCEPTION };
enum PolicyType { SecClientInvocationAccess, SecTargetInvocationAccess,
    SecApplicationAccess, SecClientInvocationAudit,
    SecTargetInvocationAudit, SecApplicationAudit, SecDelegation,
    SecClientSecureInvocation, SecTargetSecureInvocation,
    SecNonRepudiation, SecConstruction,
    PolicyType_force_long=(long)0x7fffffff };
```

Constants

```
static const int ARG_IN;
static const int ARG_OUT;
static const int ARG_INOUT;
static const int IN_COPY_VALUE;
static const int DEPENDENT_LIST;
static const int ARG_FLAGS;
static const int OBJECT NIL;
static const int OUT_LIST_MEMORY;
static const int INV_TERM_ON_ERR;
static const int RESP_NO_WAIT;
static const int INV_NO_RESPONSE;
static const int CTX_RESTRICT_SCOPE;
static const int CTX_DELETE_DESCENDENTS;
const static TypeCode_ptr _tc_null;
const static TypeCode_ptr _tc_void;
const static TypeCode_ptr _tc_short;
const static TypeCode_ptr _tc_long;
const static TypeCode_ptr _tc_ushort;
const static TypeCode_ptr _tc_ulong;
const static TypeCode_ptr _tc_float;
const static TypeCode_ptr _tc_double;
const static TypeCode_ptr _tc_boolean;
```

```

const static TypeCode_ptr _tc_char;
const static TypeCode_ptr _tc_wchar;
const static TypeCode_ptr _tc_octet;
const static TypeCode_ptr _tc_any;
const static TypeCode_ptr _tc_TypeCode;
const static TypeCode_ptr _tc_Principal;
const static TypeCode_ptr _tc_Object;
const static TypeCode_ptr _tc_string;
const static TypeCode_ptr _tc_wstring;
const static TypeCode_ptr _tc_longlong;
const static TypeCode_ptr _tc_ulonglong;
const static TypeCode_ptr _tc_CORBA_NamedValue;
const static TypeCode_ptr _tc_CORBA_InterfaceDescription;
const static TypeCode_ptr _tc_CORBA_OperationDescription;
const static TypeCode_ptr _tc_CORBA_AttributeDescription;
const static TypeCode_ptr _tc_CORBA_ParameterDescription;
const static TypeCode_ptr _tc_CORBA_ModuleDescription;
const static TypeCode_ptr _tc_CORBA_ConstantDescription;
const static TypeCode_ptr _tc_CORBA_ExceptionDescription;
const static TypeCode_ptr _tc_CORBA_TypeDescription;
const static TypeCode_ptr _tc_CORBA_FullInterfaceDescription;
static const ServiceType Security;
static const ServiceOption SecurityLevel1;
static const ServiceOption SecurityLevel2;
static const ServiceOption NonRepudiation;
static const ServiceOption SecurityORBServiceReady;
static const ServiceOption SecurityServiceReady;
static const ServiceOption ReplaceORBServices;
static const ServiceOption ReplaceSecurityServices;
static const ServiceOption StandardSecureInteroperability;
static const ServiceOption DCESecureInteroperability;
static const ServiceDetailType SecurityMechanismType;
static const ServiceDetailType SecurityAttribute;

```

CORBA::_boa

| | |
|-----------------------|--|
| Overview | Returns a pointer to the BOA object within a server. |
| Original class | "CORBA" on page 106 |

Intended Usage

Implementations of IDL interfaces, running in a server process, can use this method to obtain a BOA_ptr to the BOA object residing in the server. In the IBM implementation, _boa() is a static member function in the CORBA class, but CORBA specifies that, to be ORB-portable, applications should not assume where _boa() is defined, only that it is available within the implementation of the IDL interface. The return value should be released using CORBA::release(BOA_ptr).

IDL Syntax

```
static BOA_ptr _boa();
```

Input parameters

None

Return values

CORBA::BOA_ptr

A pointer to the BOA object residing in the server. The return value should be released using CORBA::release(BOA_ptr).

Example

See the example for the ["CORBA::is_nil" on page 106](#) method.

CORBA::is_nil

| | |
|-----------------------|---|
| Overview | Indicates whether the input object pointer represents a nil object. |
| Original class | "CORBA" on page 106 |

Intended Usage

This method is intended to be used by client and server applications to determine whether an object pointer is nil. This test should be used to verify the validity of the object prior to invoking any methods on it. This method has different signatures for different types of objects.

IDL Syntax

```
static Boolean is_nil(Any_ptr p);
static Boolean is_nil(BOA_ptr p);
static Boolean is_nil(ContextList_ptr p);
static Boolean is_nil(Context_ptr p);
static Boolean is_nil(Current_ptr p);
static Boolean is_nil(Environment_ptr p);
static Boolean is_nil(ExceptionList_ptr p);
static Boolean is_nil(Exception_ptr p);
static Boolean is_nil(NamedValue_ptr p);
static Boolean is_nil(NV_ptr p);
static Boolean is_nil(ORB_ptr p);
static Boolean is_nil(Object_ptr p);
static Boolean is_nil(Principal_ptr p);
static Boolean is_nil(Request_ptr p);
static Boolean is_nil(ServerRequest_ptr p);
static Boolean is_nil(TypeCode_ptr p);
```

Input parameters

p

The object pointer to be tested. This pointer can be NULL.

Return values

CORBA::Boolean

Returns "0" or "1". If "0" is returned, the input object pointer is valid. If "1" is returned, the input object pointer refers to a nil object.

Example

```
/* The following is a C++ example */
#include "corba.h"
...
/* Retrieve the pointer in BOA object */
CORBA::BOA_ptr pBOA;
pBOA = CORBA::_boa();
/* Test if the pointer refers to a nil object */
CORBA::Boolean bool;
bool = CORBA::is_nil(pBOA);
if (bool == TRUE)
{
    /* pBOA refers to a nil object, return or generate exception */
    ...
}
else
{
    /* proceed, using pBOA */
    ...
}
```

CORBA::ORB_init

| | |
|----------------|---|
| Overview | Initializes the ORB, if necessary, and returns a pointer to it. |
| Original class | "CORBA" on page 106 |
| Exceptions | "CORBA::SystemException" on page |

Intended Usage

This method is intended to be used by client or server applications to both initialize the ORB and obtain a pointer to it. This method can be called multiple times without adverse effect. The return value should be released using `CORBA::release(ORB_ptr)`.

Initialization of data structures used for code-set translation between clients and servers is not done until `ORB_init()` is called, so that the application has an opportunity to first initialize its locale (for instance, using the `XPG4 setlocale()` function). Therefore, if a client or server process is configured to communicate with another process using a different character code set, or is configured to use ISO-Latin 1 as the transmission code set for remote messages, the application should initialize its locale (for example, using `setlocale()`), then call `ORB_init()`, prior to using the ORB or making remote method invocations.

Note: For workstation implementations, be sure to specify `DSOM` using the parameter `argv` or `orb_identifier` as described below.

IDL Syntax

```
typedef char* ORBId;
```

```
static ORB_ptr ORB_init (int& argc, char** argv, ORBid orb_identifier);
```

Input parameters

argc

The number of strings in the *argv* array of strings. This is typically the *argc* parameter passed in to the *main()* function of the application.

argv

An array of strings, whose size is indicated by the *argc* parameter. This is typically the *argv* parameter passed in to the *main()* function of the application.

Note: For workstation implementation, if one of the strings in *argv* matches `-ORBid "DSOM"`, then ORB initialization is performed, the matching string is consumed and *argc* is decremented (the remaining strings in *argv* may be reordered as part of consuming the `-ORBid "DSOM"` string). If *argv* is NULL or contains no string that matches `-ORBid "DSOM"`, then the ORB is initialized only if the *orb_identifier* parameter is "DSOM".

orb_identifier

A string that indicates which ORB to initialize.

Note: For workstation implementation, if no string in the *argv* parameter matches `-ORBid "DSOM"`, the ORB is initialized only if the *orb_identifier* parameter is DSOM.

Return values

CORBA::ORB_ptr

A pointer to the ORB object. The return result should be released using `CORBA::release(ORB_ptr)`.

Example

```
/* The following is a C++ example */
#include "corba.h"
...
int main(int argc, char *argv[])
{
    char * orbid
    /* Initialize orbid. For CB workstation initialize to "DSOM" */
    int rc = 0;
    /* Initialize the ORB and obtain a pointer to it */
    CORBA::ORB_ptr p = CORBA::ORB_init(argc, argv, orbid);
    /* use p in the code */
    ...
    return rc;
}
```

CORBA::release

| | |
|-----------------------|--|
| Overview | Releases resources associated with an object or pseudo-object reference. |
| Original class | "CORBA" on page 106 |

Intended Usage

This method is intended to be used by client and server applications to release resources associated with object (or pseudo-object) references. `CORBA::release()` should be used regardless of whether the object is local or remote. A release does not necessarily perform a delete operation, and in general the delete operator should not be used for CORBA objects and pseudo-objects

When `CORBA::release()` is performed on a proxy to a remote implementation, the `release()` method only releases resources associated with the proxy; the remote implementation object is neither affected nor notified. When all resources associated with the proxy object are released, as determined by a reference count, the proxy object is automatically destroyed (but the remote object is unaffected). Likewise, when all local references to a local object are released, the object is automatically destroyed, regardless of how many remote (proxy) references to the object exist.

Managed objects are not destroyed when all their local references are released; instead, they are passivated and removed from memory when the Instance Manager determines the in-memory copy of the managed object is no longer needed.

The CORBA::release() method has different signatures for different types of objects and pseudo-objects.

See also the “CORBA::Object::_duplicate() method” on page 153 which is used to increase the reference count of an object reference. The _narrow() methods defined by the C++ bindings also do an implicit CORBA::Object::_duplicate().

IDL Syntax

```
static void release(BOA_ptr p);
static void release(ContextList_ptr p);
static void release(Context_ptr p);
static void release(Current_ptr p);
static void release(Environment_ptr p);
static void release(ExceptionList_ptr p);
static void release(Exception_ptr p);
static void release(NamedValue_ptr p);
static void release(NV_ptr p);
static void release(ORB_ptr p);
static void release(Object_ptr p);
static void release(Principal_ptr p);
static void release(Request_ptr p);
static void release(ServerRequest_ptr p);
static void release(TypeCode_ptr p);
```

Input parameters

p

The object reference to be released.

Return values

None.

Example

```
/* The following is a C++ example */
#include "corba.h"
#include <string.h>
...
int main(int argc, char *argv[])
{
    CORBA::Object_ptr objPtr;
    string str;
    /* Construct the string */
    CORBA::ORB_ptr op; /* assume op is initialized */
    /* Make string to object */
    objPtr = op->string_to_object(str);
    /* Proceed with objPtr */
    ...
    CORBA::string_free(string);
    CORBA::release(objPtr);
}
```

CORBA::string_alloc

| | |
|-----------------------|---------------------------------|
| Overview | Allocates storage for a string. |
| Original class | “CORBA” on page 106 |

Intended Usage

This method is intended to be used by client and server applications to dynamically allocate storage for data of type CORBA::String. The returned storage should subsequently be freed using CORBA::string_free(). Strings can also be copied using CORBA::string_dup().

Strings to be passed on remote method invocations or whose ownership is to be transferred by one library to another should be allocated using CORBA::string_alloc() (or CORBA::string_dup()) rather than the C++ new[] operator. This insures that string memory is deleted using the same C++ run time that originally allocated it.

IDL Syntax

```
static char* string_alloc(CORBA::ULong len);
```

Input parameters

len

The size of the string whose storage is to be allocated. An additional byte is also allocated for the terminating NULL character.

Return values

char*

The uninitialized string storage. This storage should later be freed using `CORBA::string_free()`. NULL is returned if the storage cannot be allocated.

Example

```
/* The following is a C++ example */
#include "corba.h"
...
/* Allocate 8 bytes for string buf */
char* buf = CORBA::string_alloc(8);
/* String copy buf */
char* buf_dup = CORBA::string_dup(buf);
/* Use buf and buf_dup */
...
/* free buf and buf_dup */
CORBA::string_free(buf);
CORBA::string_free(buf_dup);
```

CORBA::string_dup

| | |
|----------------|-------------------------------------|
| Overview | Copies a string. |
| Original class | "CORBA" on page 106 |

Intended Usage

This method is intended to be used by client and server applications to duplicate (copy) data of type `CORBA::String`. The resulting string should be subsequently freed using `CORBA::string_free()`. If the input value is NULL, the return value will be NULL.

Strings to be passed on remote method invocations or whose ownership is to be transferred from one library to another should be allocated using `CORBA::string_alloc()` or `CORBA::string_dup()` rather than the C++ `new[]` operator. This insures that string memory is deleted using the same C++ run time that originally allocated it.

IDL Syntax

```
static char* string_dup(const char* str);
```

Input parameters

str

The NULL-terminated string to be copied.

Return values

char*

A copy of the input string. This storage should be subsequently freed using `CORBA::string_free()` rather than the C++ `delete[]` operator. NULL is returned if the storage cannot be allocated.

Example

See the example for the ["CORBA::string_alloc" on page 112](#) method.

CORBA::string_free

| | |
|----------------|--|
| Overview | Frees a string allocated by <code>CORBA::string_alloc()</code> or <code>CORBA::string_dup()</code> . |
| Original class | "CORBA" on page 106 |

Intended Usage

This method is intended to be used by client and server applications to release storage

originally allocated using CORBA::string_alloc() or CORBA::string_dup().

IDL Syntax

```
static void string_free(char* str);
```

Input parameters

str

The string to be freed. If this parameter is NULL, the method has no effect.

Return values

None.

Example

See the example for the ["CORBA::string_alloc" on page 112](#) method.

CORBA::wstring_alloc

| | |
|-----------------------|-------------------------------------|
| Overview | Allocates storage for a string. |
| Original class | "CORBA" on page 106 |

Intended Usage

This method is intended to be used by client and server applications to dynamically allocate storage for data of type CORBA::String. The returned storage should subsequently be freed using CORBA::wstring_free(). Strings can also be copied using CORBA::wstring_dup().

WStrings to be passed on remote method invocations or whose ownership is to be transferred from one library to another should be allocated using this method (or CORBA::wstring_dup()) rather than the C++ new[] operator. This insures that string memory is deleted using the same C++ run time that originally allocated it.

IDL Syntax

```
static wchar_t* wstring_alloc(CORBA::ULong len);
```

Input parameters

len

The size of the string whose storage is to be allocated.

Return values

wchar_t*

The uninitialized wstring storage. This storage should later be freed using CORBA::wstring_free(). NULL is returned if the storage cannot be allocated.

Example

```
/* The following is a C++ example */
#include "corba.h"
...
/* Allocate 8 wchars for string buf */
wchar_t* buf = CORBA::wstring_alloc(8);
/* String copy buf */
wchar_t* buf_dup = CORBA::wstring_dup(buf);
/* Use buf and buf_dup */
...
/* Free buf and buf_dup */
CORBA::wstring_free(buf);
CORBA::wstring_free(buf_dup);
```

CORBA::wstring_dup

| | |
|-----------------------|-------------------------------------|
| Overview | Copies a WString. |
| Original class | "CORBA" on page 106 |

Intended Usage

This method is intended to be used by client and server applications to duplicate (copy) data

of type CORBA::WString. The resulting string should be subsequently freed using CORBA::wstring_free(). If the input value is NULL, the return value is NULL.

WStrings to be passed on remote method invocations or whose ownership is to be transferred from one library to another should be allocated using this method or CORBA::wstring_alloc() rather than the C++ new[] operator. This ensures that string memory is deleted using the same C++ run time that originally allocated it.

IDL Syntax

```
static wchar_t* * wstring_dup(const wchar_t* * str);
```

Input parameters

str

The NULL-terminated WString to be copied.

Return values

wchar_t*

A copy of the input WString. This storage should be subsequently freed using CORBA::wstring_free() rather than the C++ delete[] operator. NULL is returned if the storage cannot be allocated.

Example

See the example for the [“CORBA::wstring_alloc” on page 114](#) method.

CORBA::wstring_free

| | |
|-----------------------|--|
| Overview | Frees a WString allocated by CORBA::wstring_alloc() or CORBA::wstring_dup(). |
| Original class | “CORBA” on page 106 |
| Exceptions | “CORBA::SystemException” on page |

Intended Usage

This method is intended to be used by client and server applications to release storage originally allocated using CORBA::wstring_alloc() or CORBA::wstring_dup().

IDL Syntax

```
static void wstring_free (wchar_t * str);
```

Input parameters

str

The WString to be freed. If this parameter is NULL, the method has no effect.

Return values

None.

Example

See the example for the [“CORBA::wstring_alloc” on page 114](#) method.

CORBA module: Current Class

| | |
|--------------------------|--|
| Overview | Describes the current execution context. |
| File name | principl.h |
| Supported methods | “Current::_duplicate” on page 116 “Current::_nil” on page 116 |

Intended Usage

This abstract base class is intended to be subclassed by various object services, such as the Security Service and the Object Transaction Service. Before a Current object can be used, it must be narrowed to the appropriate subtype. Current objects are obtained using the ORB::resolve_initial_references method.

Current objects should be released using the CORBA::release(Current_ptr) method rather than the C++ delete operator.

Current::_duplicate

| | |
|----------------|--|
| Overview | Duplicates a Current object. |
| Original class | "CORBA::Current Class" on page 115 |

Intended Usage

This method is intended to be used by client and server applications to duplicate a reference to a Current object. Both the original and the duplicate reference should subsequently be released using CORBA::release(Current_ptr).

IDL Syntax

```
static CORBA::Current_ptr _duplicate (CORBA::Current_ptr p);
```

Input parameters

p

The Current object to be duplicated. The reference can be nil, in which case the return value will also be nil.

Return values

CORBA::Current_ptr

The new Current object reference. This value should subsequently be released using CORBA::release(Current_ptr).

Current::_nil

| | |
|----------------|--|
| Overview | Returns a nil CORBA::Current reference. |
| Original class | "CORBA::Current Class" on page 115 |

Intended Usage

This method is intended to be used by client and server applications to create a nil Current reference.

IDL Syntax

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

Input parameters

p

The Current object to be duplicated. The reference can be nil, in which case the return value will also be nil.

Return values

CORBA::Current_ptr

A nil Current reference.

CORBA module: EnumDef Interface

| | |
|----------|--|
| Overview | Used within the Interface Repository to represent an |
|----------|--|

| | |
|---|--|
| | OMG IDL enumeration definition. |
| File name | somir.idl |
| Local-only | True |
| Ancestor interfaces | "TypedefDef Interface" on page 228 |
| Exceptions | "CORBA::SystemException" on page |
| Supported operations | "EnumDef::members" on page 117 |
| "IDLType::type" on page 127 | |

Intended Usage

An instance of an EnumDef object is used within the Interface Repository to represent an OMG IDL enumeration definition. An instance of an EnumDef object can be created using the create_enum operation of the Container interface.

IDL syntax

```

module CORBA
{
    typedef sequence EnumMemberSeq;
    interface EnumDef:TypedefDef
    {
        attribute EnumMemberSeq members;
    };
};

```

EnumDef::members

| | |
|---------------------------|---|
| Overview | The members read and write operations provide for the access and update of the list of elements of an OMG IDL enumeration definition (EnumDef) in the Interface Repository. |
| Original interface | "CORBA module: EnumDef Interface" on page 116 |
| Exceptions | "CORBA::SystemException" on page |

Intended Usage

The members attribute contains a distinct name for each possible value of the enumeration. The members read operation provides access to a copy of the contents of this enumeration member list, and the members write operation provides the ability to update the members attribute.

IDL Syntax

```
attribute EnumMemberSeq members;
```

Read operations

Input parameters

None

Return values

CORBA::EnumMemberSeq *

The returned pointer references a sequence that is representative of the enumeration members. The memory is owned by the caller and can be released by invoking delete.

Write operations

Input parameters

CORBA::EnumMemberSeq & members

The members parameter provides the list of enumeration members with which to update the EnumDef.

Return values

None

Example

```
// C++
// assume that 'this_enum' has already been initialized
CORBA::EnumDef * this_enum;
// establish and initialize 'seq_update'
CORBA::EnumMemberSeq seq_update;
seq_update.length (3);
seq_update[0] = CORBA::string_dup ("enumerator_0");
seq_update[1] = CORBA::string_dup ("enumerator_1");
seq_update[2] = CORBA::string_dup ("enumerator_2");
// change the 'members' information in the enumeration
this_enum-> members (seq_update);
// read the 'members' information from the enumeration
CORBA::EnumMemberSeq * returned_seq;
returned_seq = this_enum-> members ();
```

CORBA module: Environment Class

| | |
|--------------------------|---|
| Overview | Holds an Exception. |
| File name | environm.h |
| Supported methods | "Environment::_duplicate" on page 118 |
| | "Environment::_nil" on page 118 |
| | "Environment::clear" on page 119 |
| | "Environment::exception" on page 119 |

Intended Usage

The Environment class holds a single Exception and is used for error handling in those cases where catch/throw exception handling cannot be used. For example, Dynamic Invocation Interface uses an Environment to report errors back to the client. The ORB::create_environment method is called to create an empty Environment. The Environment class provides methods to get and set an Exception, as well as clear an Exception. For additional information, see the Exception and Request class descriptions.

Environment::_duplicate

| | |
|-----------------------|--|
| Overview | Duplicates an Environment object. |
| Original class | "CORBA::Environment" on page 118 |

Intended Usage

This method is intended to be used by client and server applications to duplicate a reference to an Environment object. Both the original and the duplicate reference should subsequently be released using CORBA::release(Environment_ptr).

IDL Syntax

```
static CORBA::Environment_ptr _duplicate (CORBA::Environment_ptr p);
```

Input parameters

p

The Environment object to be duplicated. The reference can be nil, in which case the return value will also be nil.

Return values

CORBA::Environment_ptr

The new Environment object reference. This value should subsequently be released using CORBA::release(Environment_ptr).

Environment::_nil

| | |
|-----------------------|--|
| Overview | Returns a nil CORBA::Environment reference. |
| Original class | "CORBA::Environment" on page 118 |

Intended Usage

This method is intended to be used by client and server applications to create a nil Environment reference.

IDL Syntax

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

Input parameters

None.

Return values

CORBA::Environment_ptr

A nil Environment reference.

Environment::clear

| | |
|-----------------------|--|
| Overview | Deletes the Exception held by an Environment. |
| Original class | "CORBA::Environment" on page 118 |
| Exceptions | "CORBA::SystemException" on page |

Intended Usage

The clear method is used to delete the Exception, if any, held by an Environment.

IDL Syntax

```
void clear();
```

Input parameters

None.

Return values

None.

Environment::exception

| | |
|-----------------------|--|
| Overview | Gets and sets an Exception. |
| Original class | "CORBA::Environment" on page 118 |

Intended Usage

The exception method is used to get and set the Exception held by an Environment. The Exception returned by the get method continues to be owned by the Environment. Once the Environment is destroyed, the Exception previously returned from the get method is invalid. If the Environment does not hold an Exception, the get function returns a null pointer. The set method assumes ownership of the input Exception.

IDL Syntax

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

Input parameters

new_exception

A pointer to the new Exception to be held in the Environment. It is valid to pass a null pointer. Ownership of this parameter transfers to the Environment.

Return values

CORBA::Exception *

A pointer to the Exception currently held in the Environment, if any, or a null pointer. Ownership of the return value is maintained by the Environment; the return value must not be freed by the caller.

CORBA module: Exception Class

| | |
|--------------------------|---|
| Overview | Describes an exception condition that has occurred. |
| File name | except.h |
| Supported methods | "Exception::_duplicate" on page 120 |
| | "Exception::_nil" on page 120 |
| | "Exception::id" on page 121 |

Intended Usage

This class is intended to be caught in the catch clause of a try/catch block that encompasses remote method invocations or calls to ORB services. Typically Exception instances will actually be instances of either the SystemException or UserException subclass.

Exception::_duplicate

| | |
|-----------------------|--|
| Overview | Duplicates an Exception object. |
| Original class | "CORBA::Exception" on page 120 |

Intended Usage

This method is intended to be used by client and server applications to duplicate a reference to an Exception object. Both the original and the duplicate reference should subsequently be released using CORBA::release(Exception_ptr).

IDL Syntax

```
static CORBA::Exception_ptr _duplicate (CORBA::Exception_ptr p);
```

Input parameters

p

The Exception object to be duplicated. The reference can be nil, in which case the return value will also be nil.

Return values

CORBA::Exception_ptr

The new Exception object reference. This value should subsequently be released using CORBA::release(Exception_ptr).

Exception::_nil

| | |
|-----------------------|--|
| Overview | Returns a nil CORBA::Exception reference. |
| Original class | "CORBA::Exception" on page 120 |

Intended Usage

This method is intended to be used by client and server applications to create a nil Exception reference.

IDL Syntax

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

Input parameters

None.

Return values

CORBA::Exception_ptr

A nil Exception reference.

Exception::id

| | |
|-----------------------|--|
| Overview | Indicates the runtime type of an Exception. |
| Original class | "CORBA::Exception" on page 120 |

Intended Usage

This method is intended to be used when an Exception is caught (in the catch clause of a try/catch block), to determine the exact type of Exception that was thrown. For example, if a CORBA::NO_MEMORY exception is thrown and caught as a generic CORBA::Exception, the id() method can be invoked on the Exception, which will yield "CORBA::NO_MEMORY".

IDL Syntax

```
const char * id() const;
```

Input parameters

None.

Return values

const char *

The string name of the runtime type of the Exception object. The Exception object retains ownership of this string and the caller should not attempt to free it.

CORBA module: ExceptionDef Interface

| | |
|---|--|
| Overview | Used by the Interface Repository to represent an exception definition. |
| File name | somir.idl |
| Local-only | True |
| Ancestor interfaces | "Contained Interface" on page 79 |
| Exceptions | "CORBA::SystemException" on page |
| Supported operations | "ExceptionDef::describe" on page 122 |
| "ExceptionDef::members" on page 122 | |
| "IDLType::type" on page 127 | |

Intended Usage

The ExceptionDef object is used to represent an exception definition. An ExceptionDef object may be created in the Interface Repository database and an associated memory image of the object by calling the create_exception operation of the Container interface. The create_exception parameters include the unique RepositoryId (CORBA::RepositoryId), the name (CORBA::Identifier), the version (CORBA::VersionSpec), and a sequence indicating the exception members (CORBA::StructMemberSeq). The sequence may have zero elements to allow the ExceptionDef to have no members.

IDL syntax

```
module CORBA
{
    interface ExceptionDef:Contained
    {
```

```

        readonlyattribute TypeCode type;
        attribute StructMemberSeq members;
    };
    struct ExceptionDescription
    {
        Identifier name;
        RepositoryId id;
        RepositoryId defined_in;
        VersionSpec version;
        TypeCode type;
    };
};

```

ExceptionDef::describe

| | |
|---------------------------|---|
| Overview | Returns a structure containing information about a CORBA::ExceptionDef Interface Repository object. |
| Original interface | "CORBA module: ExceptionDef Interface" on page 121 |
| Exceptions | "CORBA::SystemException" on page |

Intended Usage

The inherited describe operation returns a structure (CORBA::Contained::Description) that contains information about a CORBA::ExceptionDef Interface Repository object. The CORBA::Contained::Description structure has two fields: kind (CORBA::DefinitionKind data type), and value (CORBA::Any data type).

The kind of definition described by the returned structure is provided using the kind field, and the value field is a CORBA::Any that contains the description that is specific to the kind of object described. When the describe operation is invoked on an exception (CORBA::ExceptionDef) object, the kind field is equal to CORBA::dk_Exception and the value field contains the CORBA::ExceptionDescription structure.

IDL Syntax

```

struct ExceptionDescription
{
    Identifier name;
    RepositoryId id;
    RepositoryId defined_in;
    VersionSpec version;
    TypeCode type;
};
struct Description
{
    DefinitionKind kind;
    any value;
};
Description describe ();

```

Input parameters

None.

Return values

Description *

The returned value is a pointer to a CORBA::Contained::Description structure. The memory is owned by the caller and can be removed by invoking delete.

Example

```

// C++
// assume that 'this_exception' has already been initialized
CORBA::ExceptionDef * this_exception;
// retrieve a description of the exception
CORBA::ExceptionDef::Description * returned_description;
returned_description = this_exception-> describe ();
// retrieve the exception description from the returned description
// structure
CORBA::ExceptionDescription * exception_description;
exception_description =
    (CORBA::ExceptionDescription *) returned_description value.value ();

```

ExceptionDef::members

| | |
|-----------------|---|
| Overview | The members read and write operations provide for the |
|-----------------|---|

| | |
|---------------------------|---|
| | access and update of the list of elements of an OMG IDL exception definition (CORBA::ExceptionDef) in the Interface Repository. |
| Original interface | "CORBA module: ExceptionDef Interface" on page 121 |
| Exceptions | "CORBA::SystemException" on page |

Intended Usage

The members attribute contains a description of each exception member. The members read and write operations allow the access and update of the members attribute.

IDL Syntax

```
attribute StructMemberSeq members;
```

Read operations

Input parameters

None.

Return values

CORBA::StructMemberSeq *

The returned pointer references a sequence that is representative of the exception members. The memory is owned by the caller and can be released by invoking delete.

Write operations

Input parameters

CORBA::StructMemberSeq & members

The members parameter provides the list of exception members with which to update the ExceptionDef. Setting the members attribute also updates the type attribute.

Return values

None.

Example

```
// C++
// assume 'this_exception_def', 'struct_1', and 'struct_2'
// have already been initialized
CORBA::ExceptionDef * this_exception_def;
CORBA::StructDef * struct_1;
CORBA::StructDef * struct_2;
// establish and initialize the StructMemberSeq . . .
CORBA::StructMemberSeq seq_update;
seq_update.length (2);
seq_update[0].name = CORBA::string_dup ("struct_1");
seq_update[0].type_def = CORBA::IDLType::_duplicate (struct_1);
seq_update[1].name = CORBA::string_dup ("struct_2");
seq_update[1].type_def = CORBA::IDLType::_duplicate (struct_2);
// set the members attribute of the ExceptionDef
this_exception_def-> members (seq_update);
// read the members attribute information from the ExceptionDef
CORBA::StructMemberSeq * returned_members;
returned_members = this_exception_def-> members ();
```

CORBA module: ExceptionList Class

| | |
|--------------------------|--|
| Overview | Specifies the list of user-defined exceptions that can be raised when a DII request is executed. |
| File name | excp_lst.h |
| Supported methods | "ExceptionList::_duplicate" on page 124 |
| | "ExceptionList::_nil" on page 124 |
| | "ExceptionList::add" on page 125 |
| | "ExceptionList::add_consume" on page 125 |
| | "ExceptionList::count" on page 125 |

["ExceptionList::item" on page 126](#)

["ExceptionList::remove" on page 126](#)

Intended Usage

When a client assembles a Dynamic Invocation Interface request, an `ExceptionList` is optionally included. An `ExceptionList` specifies the list of `TypeCodes` for all user-defined exceptions that can be raised when a request is executed. The `ExceptionList` class is used to improve performance. When invoking a request without an `ExceptionList`, the ORB looks up user-defined exception information in the Interface Repository. The `ORB::create_exception_list` method is called to create an empty exception list. The `ExceptionList` class provides methods to add and delete an exception, as well as query information about an exception list. For additional information, see the `Exception`, `UserException`, and `Request` class descriptions.

ExceptionList::_duplicate

| | |
|----------------|--|
| Overview | Duplicates an <code>ExceptionList</code> object. |
| Original class | "CORBA::ExceptionList" on page 123 |

Intended Usage

This method is intended to be used by client and server applications to duplicate a reference to an `ExceptionList` object. Both the original and the duplicate reference should subsequently be released using `CORBA::release(ExceptionList_ptr)`.

IDL Syntax

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

Input parameters

p

The `ExceptionList` object to be duplicated. The reference can be `nil`, in which case the return value will also be `nil`.

Return values

CORBA::ExceptionList_ptr

The new `ExceptionList` object reference. This value should subsequently be released using `CORBA::release(ExceptionList_ptr)`.

ExceptionList::_nil

| | |
|----------------|---|
| Overview | Returns a <code>nil</code> <code>CORBA::ExceptionList</code> reference. |
| Original class | "CORBA::ExceptionList" on page 123 |

Intended Usage

This method is intended to be used by client and server applications to create a `nil` `ExceptionList` reference.

IDL Syntax

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

Input parameters

None.

Return values

CORBA::ExceptionList_ptr

A `nil` `ExceptionList` reference.

ExceptionList::add

| | |
|-----------------------|--|
| Overview | Adds a single user-defined exception to an exception list. |
| Original class | "CORBA::ExceptionList" on page 123 |
| Exceptions | "CORBA::SystemException" on page |

Intended Usage

The add method is used by a client program to populate the ExceptionList associated with a DII request. The add method adds a single user-defined exception to an exception list. The add and add_consume methods perform the same task but differ in memory management. The add method does not assume ownership of the input TypeCode; the add_consume method does.

IDL Syntax

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

Input parameters

tc

A pointer to the TypeCode for the user-defined exception. A system exception is raised if the input pointer is null.

Return values

None.

ExceptionList::add_consume

| | |
|-----------------------|--|
| Overview | Adds a single user-defined exception to an exception list. |
| Original class | "CORBA::ExceptionList" on page 123 |
| Exceptions | "CORBA::SystemException" on page |

Intended Usage

The add_consume method is used by a client program to populate the ExceptionList associated with a DII request. The add_consume method adds a single user-defined exception to an exception list. The add and add_consume methods perform the same task but differ in memory management. The add_consume method assumes ownership of the input TypeCode; the add method does not. The caller must not access the object referred to by the input parameter after it has been passed in.

IDL Syntax

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

Input parameters

tc

A pointer to the TypeCode for the user-defined exception. The input TypeCode must either be retrieved from the Interface Repository or allocated using the ORB::create_exception_tc method. Ownership of this parameter transfers to the ExceptionList. A system exception is raised if the input pointer to the TypeCode is null.

Return values

None.

ExceptionList::count

| | |
|-----------------|--|
| Overview | Retrieves the number of elements in an exception list. |
|-----------------|--|

| | |
|----------------|--|
| Original class | "CORBA::ExceptionList" on page 123 |
|----------------|--|

Intended Usage

The count method is used by a client program when querying the ExceptionList associated with a DII request. The count method returns the number of elements in an exception list.

IDL Syntax

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

Input parameters

None.

Return values

CORBA::ULong

The number of elements in the exception list.

ExceptionList::item

| | |
|----------------|--|
| Overview | Retrieves the user-defined exception associated with an input index. |
| Original class | "CORBA::ExceptionList" on page 123 |
| Exceptions | "CORBA::SystemException" on page |

Intended Usage

The item method is used by a client program when querying the ExceptionList associated with a DII request. The item method returns the user-defined exception associated with an input index.

IDL Syntax

```
CORBA::TypeCode_ptr item(CORBA::ULong index)
```

Input parameters

index

The index of the desired user-defined exception, starting at zero. A system exception is raised if the input index is greater than or equal to the number of elements in the exception list.

Return values

CORBA::TypeCode_ptr

A pointer to the TypeCode for the user-defined exception. Ownership of the return value is maintained by the ExceptionList; the return value must not be freed by the caller.

ExceptionList::remove

| | |
|----------------|--|
| Overview | Deletes the user-defined exception associated with an input index. |
| Original class | "CORBA::ExceptionList" on page 123 |
| Exceptions | "CORBA::SystemException" on page |

Intended Usage

The remove method is used by a client program when deleting an element of the ExceptionList associated with a DII request. The remove method deletes the user-defined exception associated with an input index. The remaining exceptions are re-indexed.

IDL Syntax

```
CORBA::Status remove(CORBA::ULong index);
```

Input parameters

index

The index of the user-defined exception to be deleted, starting at zero. A system exception is raised if the input index is greater than or equal to the number of elements in the exception list.

Return values

CORBA::Status

A zero return code indicates the user-defined exception was successfully deleted.

CORBA module: IDLType Interface

| | |
|-----------------------------|---|
| Overview | An abstract interface inherited by all Interface Repository objects that represent OMG IDL types. It provides access to the TypeCode that describes the type. The IDL Type is used in defining other interfaces whenever definitions of IDL types must be referenced. |
| File name | somir.idl |
| Local-only | True |
| Ancestor interfaces | "IObject Interface" on page 138 |
| Exceptions | "CORBA::SystemException" on page |
| Supported operations | "IDLType::type" on page 127 |

Intended Usage

The IDLType interface is not itself instantiated as a means of accessing the Interface Repository. As an ancestor to Interface Repository objects that represent OMG IDL types, it provides a specific operation as noted below. Those Interface Repository objects that inherit (directly or indirectly) the operation defined in IDLType include: StructDef, UnionDef, EnumDef, AliasDef, PrimitiveDef, StringDef, WstringDef, SequenceDef, ArrayDef, and InterfaceDef.

IDL syntax

```
module CORBA
{
    interface IDLType:IObject
    {
        readonly attribute TypeCode type;
    };
};
```

IDLType::type

| | |
|---------------------------|--|
| Overview | The type operation retrieves a TypeCode pointer representative of specific Interface Repository objects. |
| Original interface | "CORBA module: IDLType Interface" on page 127 |
| Exceptions | "CORBA::SystemException" on page |

Intended Usage

The type attribute (a TypeCode *) describes all objects derived from IDLType. The type read operation retrieves a pointer to a copy of the type attribute. Object types that inherit from IDLType and therefore support the type read operation are ArrayDef, SequenceDef, StringDef, WstringDef, PrimitiveDef, UnionDef, StructDef, AliasDef, EnumDef, and InterfaceDef.

There are other Interface Repository objects that do not inherit from IDLType that also have

a type method that returns a `TypeCode *` representative of the specific object. The Interface Repository interfaces that have their own type method include: `ConstantDef`, `ExceptionDef`, and `AttributeDef`.

IDL Syntax

```
readonly attribute TypeCode type;
```

Input parameters

None.

Return values

`TypeCode_ptr`

The return value is a pointer to a `TypeCode` that describes the object. The memory associated with the returned `TypeCode` pointer is owned by the caller and can be released by calling `CORBA::release`.

Example

```
// C++
// assume that 'union_1' has already been initialized
CORBA::UnionDef * union_1;
// retrieve the TypeCode information which represents 'union_1' . . .
CORBA::TypeCode * typecode_ptr;
typecode_ptr = union_1-> type();
```

CORBA module: `ImplementationDef` Interface

| | |
|-----------------------------|---|
| Overview | Describes a logical server as registered in the Implementation Repository. |
| File name | impldef.h |
| Supported operations | "ImplementationDef::get_alias" on page 129 "ImplementationDef::get_id" on page 129 |

Intended Usage

`CORBA::ImplementationDef` objects represent logical server applications. They are stored persistently in the Implementation Repository, represented programmatically by the `CORBA::ImplRepository` class. `ImplementationDef` objects are stored and updated in the Implementation Repository as servers are registered, unregistered, or changed. Typically this administration of the Implementation Repository is done using the product tools, but it can also be done programmatically (using the `ImplementationDef` and `ImplRepository` classes).

The `CORBA::ImplementationDef` class is used in the following ways:

- By the `somorbd` daemon, to find and activate servers;
- By server applications, to initialize themselves (the `BOA::impl_is_ready` and `BOA::deactivate_impl` methods require an `ImplementationDef` parameter);
- By applications written to programmatically query or update the contents of the Implementation Repository (typically this is done using the product tools);
- By client applications, to discover information about servers with which they are communicating, using the `CORBA::Object::_get_implementation` method.

`ImplementationDef` objects contain the following data to describe registered servers:

- Server ID (a UUID that uniquely identifies the server throughout a network and is used as a key into the Implementation Repository),
- Server alias (a user-friendly, administrator-defined name that uniquely identifies the

server on a given machine, but not necessarily throughout the network, and can be used as a key into the Implementation Repository),

- Server program name (the executable that implements the logical server, which the somorbd daemon starts to activate the server on demand; this need not be a fully-qualified pathname, and needn't be unique to a particular server),
- The communication protocol(s) that the server supports (e.g., SOMD_TCPIP, SOMD_IPC),
- The key to the server's configuration data, if different from the somorbd daemon's (set by the somorbd daemon before starting the server),
- Flag bits, defined and used by the ORB.

In addition, applications can store arbitrary name/value pairs in ImplementationDef objects; these values can be used by the application to control server behavior.

Types

```
typedef sequence<nameValue> seq_nameValue;
typedef sequence<string> seq_string;
struct nameValue {
    string name;
    string value;
};
```

ImplementationDef::get_alias

| | |
|---------------------------|---|
| Overview | Retrieves the user-defined alias of the logical server represented by an ImplementationDef. |
| Original interface | "CORBA::ImplementationDef" on page 128 |
| Exceptions | "CORBA::SystemException" on page |

Intended Usage

This method is intended to be used by an application to retrieve the user-friendly, administrator-assigned name (alias) of a logical server registered in the Implementation Repository.

Note: This is an IBM-defined operation, which extends the CORBA 2.1 specifications provided by the OMG.

IDL Syntax

```
string get_id();
```

Input parameters

None.

Return values

string

The alias of the logical server represented by the ImplementationDef. The caller assumes ownership of the returned string, and should subsequently deallocate it using the CORBA::string_free function.

ImplementationDef::get_id

| | |
|---------------------------|--|
| Overview | Retrieves the ORB-assigned UUID of the logical server represented by an ImplementationDef. |
| Original interface | "CORBA::ImplementationDef" on page 128 |
| Exceptions | "CORBA::SystemException" on page |

Intended Usage

This method is intended to be used by an application to retrieve the ORB-assigned UUID of

a logical server registered in the Implementation Repository. This id is unique throughout the network and can be used as a key into the Implementation Repository.

Note: This is an IBM-defined operation, which extends the CORBA 2.1 specifications provided by the OMG.

IDL Syntax

```
string get_id();
```

Input parameters

None.

Return values

string

The UUID (id) of the logical server represented by the ImplementationDef. The caller assumes ownership of the returned string, and should subsequently deallocate it using the CORBA::string_free function.

CORBA module: ImplRepository Class

| | |
|--------------------------|---|
| Overview | Represents a persistent data store of ImplementationDef objects, each representing a logical server that has been registered. |
| File name | implrep |
| Supported methods | "ImplRepository::find_impldef" on page 130 "ImplRepository::find_impldef_by_alias" on page 131 |

Intended Usage

The ImplRepository class represents the Implementation Repository, a persistent data store of ImplementationDef objects, each representing a logical server that has been registered.

The ImplRepository class is intended to be used by server applications, using the find_impldef or find_impldef_by_alias method, to retrieve the ImplementationDef representing that server application. Each server must retrieve its own ImplementationDef object from the Implementation Repository (using the ImplRepository class), because the ImplementationDef is a parameter required by the BOA::impl_is_ready method.

The other methods of the ImplRepository class are not typically used by client or server applications, but can be used by an application to programmatically administer the Implementation Repository. (Typically servers are registered and unregistered from the Implementation Repository using the product tools, rather than programmatically by an application.)

ImplRepository::find_impldef

| | |
|-----------------------|--|
| Overview | Retrieves an ImplementationDef from the Implementation Repository, by server id. |
| Original class | "CORBA::ImplRepository" on page 130 |
| Exceptions | "CORBA::SystemException" on page |

Intended Usage

This method is used to retrieve an ImplementationDef object from the Implementation Repository, using the server id as a key. This method (or the find_impldef_by_alias method)

is intended to be used by a server application to retrieve its own ImplementationDef object from the Implementation Repository; a server application then passes this ImplementationDef object to the BOA::impl_is_ready and BOA::deactivate_impl methods.

A CORBA::SystemException is thrown if the Implementation Repository cannot be accessed (for instance, due to a configuration error), if the input server id is NULL or is not in the correct format, or if the specified server id cannot be found in the Implementation Repository.

Note: This is an IBM-defined operation, which extends the CORBA 2.0 specifications provided by the OMG.

IDL Syntax

```
CORBA::ImplementationDef* find_impldef(const char * ImplId);
```

Input parameters

ImplId

The server id of the ImplementationDef to be retrieved. A CORBA::SystemException will be thrown if this string is NULL or is not in the proper format of an Implementation Repository UUID (as created by the ImplementationDef constructor).

Return values

CORBA::ImplementationDef*

The CORBA::ImplementationDef from the Implementation Repository whose id matches the input. The caller assumes ownership of this object and must subsequently delete it. (If passed to BOA::impl_is_ready, however, the ImplementationDef object should not be deleted until after BOA::deactivate_impl has subsequently been called and the server has quiesced, to insure that the BOA does not subsequently refer to that object.) Each call to find_impldef returns a different ImplementationDef object (although the different objects will be equivalent for equivalent input to find_impldef).

ImplRepository::find_impldef_by_alias

| | |
|-----------------------|---|
| Overview | Retrieves an ImplementationDef from the Implementation Repository, by server alias. |
| Original class | "CORBA::ImplRepository" on page 130 |
| Exceptions | "CORBA::SystemException" on page |

Intended Usage

This method is used to retrieve an ImplementationDef object from the Implementation Repository, using the server alias as a key. This method (or the find_impldef method) is intended to be used by a server application to retrieve its own ImplementationDef object from the Implementation Repository; a server application then passes this ImplementationDef object to the BOA::impl_is_ready and BOA::deactivate_impl methods.

A CORBA::SystemException is thrown if the Implementation Repository cannot be accessed (for instance, due to a configuration error), if the input server alias is NULL, or if the specified server alias cannot be found in the Implementation Repository.

Note: This is an IBM-defined operation, which extends the CORBA 2.1 specifications provided by the OMG.

IDL Syntax

```
CORBA::ImplementationDef* find_impldef(const char * ImplAlias);
```

Input parameters

ImplAlias

The server alias of the ImplementationDef to be retrieved. A CORBA::SystemException will be thrown if this string is NULL.

Return values

CORBA::ImplementationDef*

The CORBA::ImplementationDef from the Implementation Repository whose alias matches the input. The caller assumes ownership of this object and must subsequently delete it. (If passed to BOA::impl_is_ready, however, the ImplementationDef object should not be deleted until after BOA::deactivate_impl has subsequently been called and the server has quiesced, to insure that the BOA does not subsequently refer to that object.) Each call to find_impldef_by_alias returns a different ImplementationDef object (although the different objects will be equivalent for equivalent input to find_impldef_by_alias).

CORBA module: InterfaceDef Interface

| | |
|-----------------------------|---|
| Overview | The InterfaceDef object represents an interface definition in the Interface Repository. |
| File name | somir.idl |
| Local-only | True |
| Ancestor interfaces | "Contained Interface" on page 79 "Container Interface" on page 85 "IDLType Interface" on page 127 |
| Exceptions | "CORBA::SystemException" on page |
| Supported operations | "InterfaceDef::base_interfaces" on page 133 "InterfaceDef::create_attribute" on page 134 "InterfaceDef::create_operation" on page 135 "InterfaceDef::describe" on page 136 "InterfaceDef::describe_interface" on page 137 "InterfaceDef::is_a" on page 138 |

Intended Usage

The InterfaceDef object is used to represent an interface definition. An InterfaceDef object may be created in the Interface Repository database and an associated memory image of the object by calling the create_interface operation of the Container interface. The create_interface parameters include the unique RepositoryId (CORBA::RepositoryId), the name (CORBA::Identifier), the version (CORBA::VersionSpec), and a sequence indicating the base interfaces from which the interface inherits.

IDL syntax

```

module CORBA
{
    interface InterfaceDef;
    typedef sequence InterfaceDefSeq;
    typedef sequence RepositoryIdSeq;
    typedef sequence OpDescriptionSeq;
    typedef sequence AttrDescriptionSeq;
    Interface InterfaceDef: Container, Contained, IDLType
    {
        //read/write interface
        attribute InterfaceDefSeq base_interfaces;
        //read interface
        boolean is_a(in RepositoryId interface_id);
        struct FullInterfaceDescription
        {
            Identifier name;
            Repository Id id;
            RepositoryId defined_in;
            VersionSpec version;
        };
    };
};

```

```

OpDescriptionSeq operations;
AttrDescriptionSeq attributes;
RepositoryIdSeq base_interfaces;
TypeCode type;
};
FullInterfaceDescription describe_interface();
// write interface
AttributeDef create_attribute (in RepositoryId id,
                               in Identifier name,
                               in VersionSpec version,
                               in IDLType type,
                               in AttributeMode mode);
OperationDef create_operation (in RepositoryId id,
                               in Identifier name,
                               in VersionSpec version,
                               in IDLType result,
                               in OperationMode mode,
                               In ParDescriptionSeq params,
                               In ExceptionDefSeq exceptions,
                               in ContextIdSeq contexts);
};
struct InterfaceDescription
{
    Identifier name;
    RepositoryId id;
    RepositoryID defined_in;
    VersionSpec version;
    RepositoryId Seqbase_interfaces;
};

```

InterfaceDef::base_interfaces

| | |
|---------------------------|---|
| Overview | The base_interface attribute is a list of all the interfaces from which the current interface directly inherits. The base_interface operations read/write the base_interface attribute of the target interface. |
| Original interface | "InterfaceDef Interface" on page 132 |
| Exceptions | "CORBA::SystemException" on page |

Intended Usage

Both Read and Write methods are supported, with parameters given below.

IDL Syntax

```
attribute InterfaceDefSeq base_interfaces;
```

Read operations

Input parameters

None.

Return values

CORBA::InterfaceDefSeq *

A pointer is returned to a copy of the base_interfaces attribute. The memory associated with the returned value is owned by the caller and may be released by invoking CORBA::delete.

Write operations

Input parameters

CORBA::InterfaceDefSeq

The sequence defines the new list of InterfaceDefs from which the target interface will be changed to inherit.

Return values

None.

Example

```

// C++
// assume 'interface_1', 'interface_99', and 'interface_100'
// have already been initialized . . .
CORBA::InterfaceDef * interface_1;
CORBA::InterfaceDef * interface_99;
CORBA::InterfaceDef * interface_100;
// establish a new list for the interface_1 inheritance
CORBA::InterfaceDefSeq new_base_interfaces;
new_base_interfaces.length (2);

```

```

new_base_interfaces[0] = CORBA::InterfaceDef::_duplicate (interface_99);
new_base_interfaces[1] = CORBA::InterfaceDef::_duplicate (interface_100);
// change the base interfaces for interface_1
interface_1-> base_interfaces (new_base_interfaces);
// retrieve the 'base interfaces' attribute
CORBA::InterfaceDefSeq * returned_base_interfaces;
returned_base_interfaces = interface_1-> base_interfaces ();

```

InterfaceDef::create_attribute

| | |
|---------------------------|---|
| Overview | The create_attribute operation adds a new attribute definition to an interface definition on which it is invoked. |
| Original interface | "InterfaceDef Interface" on page 132 |
| Exceptions | "CORBA::SystemException" on page |

Intended Usage

The create_attribute operation adds a new CORBA::AttributeDef contained in the CORBA::InterfaceDef on which it is invoked. A representation of the new CORBA::AttributeDef object is created in the Interface Repository persistent database and a pointer to the memory representation of the CORBA::AttributeDef object is returned to the caller.

The id, name, version, typedef, and mode attributes are set as specified. The type attribute is also set. The defined_in attribute is initialized to identify the containing CORBA::InterfaceDef.

An error is returned if an object with the specified id already exists within this object's CORBA::RepositoryId, or if an object with the specified name already exists within this CORBA::InterfaceDef.

IDL Syntax

```

AttributeDef create_attribute (in RepositoryId id,
                             in Identifier name,
                             in VersionSpec version,
                             in IDLType type,
                             in AttributeMode mode);

```

Input parameters

mode

Valid attribute mode values are CORBA::ATTR_NORMAL (read/write access) and CORBA::ATTR_READONLY (read only access).

name

The name that will be associated with this CORBA::AttributeDef object in the Interface Repository.

type_def

The type_def parameter is a CORBA::IDLType pointer that specifies the type of the CORBA::AttributeDef.

id

The id represents the CORBA::RepositoryId that will uniquely identify this CORBA::AttributeDef within the Interface Repository.

version

The version number that will be associated with this CORBA::AttributeDef object in the Interface Repository.

Return values

AttributeDef *

The returned value is a pointer to the created CORBA::AttributeDef object. The memory is owned by the caller and may be released using CORBA::release.

Example

```
// C++
// assume 'this_interface' and 'pk_long_ptr'
// have already been initialized
CORBA::InterfaceDef * this_interface;
CORBA::PrimitiveDef * pk_long_ptr;
// establish the 'create_attribute' parameters
CORBA::RepositoryId rep_id = CORBA::string_dup ("UniqueRepositoryId");
CORBA::Identifier name = CORBA::string_dup ("this_attribute");
CORBA::VersionSpec version = CORBA::string_dup ("1.0");
CORBA::AttributeMode mode = CORBA::ATTR_READONLY;
// create the new attribute definition contained in the interface
CORBA::AttributeDef * this_attribute;
this_attribute = this_interface->create_attribute(rep_id, name, version,
                                                pk_long_ptr, mode);
```

InterfaceDef::create_operation

| | |
|---------------------------|---|
| Overview | The create_operation operation returns a new operation definition (CORBA::OperationDef) contained in the interface definition (CORBA::InterfaceDef) on which it is invoked. |
| Original interface | "InterfaceDef Interface" on page 132 |
| Exceptions | "CORBA::SystemException" on page |

Intended Usage

The create_operation operation creates a new OperationDef in the Interface Repository and returns a pointer to the memory representation of the CORBA::OperationDef object. The id, name, version, result_def, mode, params, exceptions, and contexts attributes are set as specified. The result attribute is also set based on result_def. The defined_in attribute is initialized to identify the containing CORBA::InterfaceDef.

An error is returned if an object with the specified id already exists within the Interface Repository or if an object with the specified name already exists within this CORBA::InterfaceDef.

IDL Syntax

```
OperationDef create_operation (in RepositoryId id,
                              in Identifier name,
                              in VersionSpec version,
                              in IDLType result,
                              in OperationMode mode,
                              in ParDescriptionSeq params,
                              in ExceptionDefSeq exceptions,
                              in ContextIdSeq contexts);
```

Input parameters

id

The id represents the CORBA::RepositoryId that will uniquely identify this CORBA::OperationDef within the Interface Repository.

name

The name that will be associated with this CORBA::OperationDef object in the Interface Repository.

version

The version number that will be associated with this CORBA::OperationDef object in the Interface Repository.

result_def

The result_def parameter is a CORBA::IDLType * that specifies the type of the return value for the CORBA::OperationDef.

mode

Valid operation mode values include CORBA::OP_NORMAL (normal operation) and CORBA::OP_ONEWAY (one way operation).

params

The sequence defines the list of parameters that are associated with the interface.

exceptions

The sequence defined the list of exceptions that are associated with the interface.

contexts

The sequence defines the list of contexts that are associated with the interface.

Return values

OperationDef *

The returned value is a pointer to the created CORBA::OperationDef object. The memory is owned by the caller and can be released using CORBA::release.

Example

```
// C++
// assume 'this_interface', 'this_struct', 'this_exception', and
// assume 'this_interface', 'this_struct', 'this_exception', and
// 'pk_long_ptr' have already been defined
CORBA::InterfaceDef * this_interface;
CORBA::StructDef * this_struct;
CORBA::ExceptionDef * this_exception;
CORBA::PrimitiveDef * pk_long_ptr;
// establish the 'create_operation' parameters
CORBA::RepositoryId rep_id = CORBA::string_dup ("UniqueRepositoryId");
CORBA::Identifier name = CORBA::string_dup ("this_operation");
CORBA::VersionSpec version = CORBA::string_dup ("1.0");
CORBA::IDLType * result_def = this_struct;
CORBA::OperationMode mode = CORBA::OP_NORMAL;
CORBA::ParDescriptionSeq params;
params.length (1);
params[0].name = CORBA::string_dup ("parameter_0");
params[0].type = CORBA::tc_long;
params[0].type_def = pk_long_ptr;
params[0].mode = CORBA::PARAM_IN;
CORBA::ExceptionDefSeq exceptions;
exceptions.length (1);
exceptions[0] = this_exception;
CORBA::ContextIdSeq contexts;
contexts.length (1);
contexts[0] = CORBA::string_dup ("CONTEXTS_0=value_0");
// create the operation . . .
CORBA::OperationDef * this_operation;
this_operation = this_interface-> create_operation
(rep_id, name, version, result_def, mode, params, exceptions, contexts);
```

InterfaceDef::describe

| | |
|---------------------------|--|
| Overview | The describe operation returns a structure containing information about an InterfaceDef Interface Repository object. |
| Original interface | "InterfaceDef Interface" on page 132 |
| Exceptions | "CORBA::SystemException" on page |

Intended Usage

The inherited describe operation returns a structure (CORBA::Contained::Description) that contains information about a CORBA::InterfaceDef Interface Repository object. The CORBA::Contained::Description structure has two fields: kind (CORBA::DefinitionKind data type), and value (CORBA::Any data type).

The kind of definition described by the returned structure is provided by invoking the kind field, and the value field is a CORBA::Any containing the description specific to the kind of object described. When the describe operation is invoked on an interface (CORBA::InterfaceDef) object, the kind field is equal to CORBA::dk_Interface and the value field contains the CORBA::InterfaceDescription structure.

IDL Syntax

```
struct InterfaceDescription
{
    Identifier name;
    RepositoryId id;
    RepositoryId defined_in;
    VersionSpec version;
    RepositoryIdSeq base_interfaces;
};
struct Description
{
    DefinitionKind kind;
    any value;
```

```
};
Description describe ();
```

Input parameters

None.

Return values

Description *

The returned value is a pointer to a CORBA::Contained::Description structure. The memory is owned by the caller and can be removed by invoking delete.

Example

```
// C++
// assume that 'this_interface' has already been initialized
CORBA::InterfaceDef * this_interface;
// retrieve a description of the interface
CORBA::Contained::Description * returned_description;
returned_description = this_interface-> describe ();
// retrieve the interface description from the returned description
// structure
CORBA::InterfaceDescription * interface_description;
interface_description = (CORBA::InterfaceDescription *)
    returned_description value.value ();
```

InterfaceDef::describe_interface

| | |
|---------------------------|---|
| Overview | The describe_interface operation returns a CORBA::InterfaceDef::FullInterfaceDescription describing the interface (CORBA::InterfaceDef), including its operations and attributes. |
| Original interface | "CORBA::InterfaceDef" on page 132 |
| Exceptions | "CORBA::SystemException" on page |

Intended Usage

The data fields of the CORBA::InterfaceDef::FullInterfaceDescription include: name (the interface name), id (the unique CORBA::RepositoryId that identifies the interface), defined_in (the unique CORBA::RepositoryId that identifies the defined_in attribute), version (the version number), operations (a sequence listing the operations for this interface), attributes (a sequence listing the attributes of this interface), base_interfaces (a sequence of CORBA::RepositoryIds that represent the base_interfaces attribute of the interface), and type (the CORBA::TypeCode representation of the interface).

Note: The CORBA specification contains ambiguities with relation to the describe_interface. This method returns the FullInterfaceDescription structure that contains the interface's attributes and operations. It does not state whether that includes or excludes the inherited attributes and operations. The IBM implementation excludes the inherited attributes and operations.

IDL Syntax

```
struct FullInterfaceDescription
{
    Identifier name;
    RepositoryId id;
    RepositoryId defined_in;
    VersionSpec version;
    OpDescriptionSeq operations;
    AttrDescriptionSeq attributes;
    RepositoryIdSeq base_interfaces;
    TypeCode type;
};
FullInterfaceDescription describe_interface();
```

Input parameters

None.

Return values

FullInterfaceDescription *

The returned value is owned by the caller and can be removed by invoking delete.

Example

```
// C++
// assume that 'this_interface' has already been initialized
CORBA::InterfaceDef * this_interface;
// retrieve a full description of the interface
CORBA::InterfaceDef::FullInterfaceDescription *
returned_full_interface_description;
returned_full_interface_description = this_interface->
describe_interface ();
```

InterfaceDef::is_a

| | |
|---------------------------|--|
| Overview | The is_a operation is used to determine if the target interface is identical to or inherits from another interface referenced by its unique CORBA::RepositoryId. |
| Original interface | "CORBA::InterfaceDef" on page 132 |
| Exceptions | "CORBA::SystemException" on page |

Intended Usage

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 is_a read operation parameter and result description is provided below.

IDL Syntax

```
boolean is_a(in RepositoryId interface_id);
```

Input parameters

interface_id

The ID attribute that globally identifies a Contained object.

Return values

Boolean

The return value is the result of the evaluation of the target object and the referenced object as in the Intended Usage section.

Example

```
// C++
// assume 'this_interface' and 'other_interfaces_rep_id'
// have already been initialized
CORBA::InterfaceDef * this_interface;
CORBA::RepositoryId other_interfaces_rep_id;
// determine if the two objects are related
CORBA::Boolean returned_boolean;
returned_boolean = this_interface-> is_a (other_interfaces_rep_id);
```

CORBA module: IRObect Interface

| | |
|--|--|
| Overview | The IRObect interface represents the most generic interface from which all other Interface Repository interfaces are derived, including the Repository itself. |
| File name | somir.idl |
| Local-only | True |
| Ancestor interfaces | None |
| Exceptions | "CORBA::SystemException" on page |
| Supported operations | "IRObect::def_kind" on page 139 |
| "IRObect::destroy" on page 139 | |

Intended Usage

The IRObect is not itself instantiated as a means of accessing the Interface Repository. As

an ancestor of all Interface Repository objects, it defines the specific operations noted above.

IDL syntax

```
module CORBA {
  interface IObject {
    //read Interface
    read only attribute DefinitionKind def_kind;
    //write interface
    void destroy ();
  }
}
```

Types

```
enum DefinitionKind
{dk_none, dk_all, dk_Attribute, dk_Constant, dk_Exception, dk_Interface,
  dk_Module, dk_Operation, dk_Typedef, dk_Alias, dk_Struct, dk_Union,
  dk_Enum, dk_Primitive, dk_String, dk_Sequence, dk_Array, dk_Repository
};
```

IObject::def_kind

| | |
|---------------------------|---|
| Overview | The def_kind operation returns the kind of the Interface Repository definition. |
| Original interface | "IObject Interface" on page 138 |
| Exceptions | "CORBA::SystemException" on page |

Intended Usage

The def_kind attribute identifies the kind of the Interface Repository definition. The def_kind operation returns the value in this attribute that identifies the definition kind of the object.

IDL Syntax

```
readonly attribute DefinitionKind def_kind;
```

Input parameters

None.

Return values

DefinitionKind

The returned value indicates the definition kind of the Interface Repository object. Valid values that are returned by the def_kind read operation include: CORBA::dk_Attribute, CORBA::dk_Constant, CORBA::dk_Exception, CORBA::dk_Interface, CORBA::dk_Module, CORBA::dk_Operation, CORBA::dk_Alias, CORBA::dk_Struct, CORBA::dk_Union, CORBA::dk_Enum, CORBA::dk_Primitive, CORBA::dk_String, CORBA::dk_Sequence, CORBA::dk_Array, and CORBA::dk_Repository.

Example

```
// C++
// assume 'ir_object_ptr' has already been initialized . . .
CORBA::IObject * ir_object_ptr;
// query the object to determine the definition kind . . .
CORBA::DefinitionKind this_objects_kind;
this_objects_kind = ir_object_ptr-> def_kind();
```

IObject::destroy

| | |
|---------------------------|---|
| Overview | The destroy operation causes the object to cease to exist within the Interface Repository database. |
| Original interface | "IObject Interface" on page 138 |
| Exceptions | "CORBA::SystemException" on page |

Intended Usage

The destroy operation causes the object to cease to exist. If the object is a Container, destroy is applied to all of its contents. If the object contains an IDLType attribute for an

anonymous type, that IDLType is destroyed. If the object is currently contained in some other object, it is removed from that container object. Invoking destroy on a Repository object or on a PrimitiveDef is an error.

The destroy operation causes the object to cease to exist. If the object is a Container, destroy is applied to all of its contents. If the object contains an IDLType attribute for an anonymous type, that IDLType is destroyed. If the object is currently contained in some other object, it is removed from that container object. Invoking destroy on a Repository object or on a PrimitiveDef is an error. CORBA 2.1 requires that the IR not be left in an incoherent state. After a destroy there cannot be any dangling references. The IBM implementation of destroy ensures this by deleting all objects that refer to the destroy target. When destroying an interface this will include all of its children. Use caution.

IDL Syntax

```
void destroy ();
```

Input parameters

None.

Return values

Void

No value is returned.

Example

```
// C++
// assume that 'this_module' has already been initialized
CORBA::ModuleDef * this_module;
// destroy the module and all that it contains
this_module-> destroy ();
```

CORBA module: ModuleDef Interface

| | |
|-----------------------------|--|
| Overview | A ModuleDef can contain constants, typedefs, exceptions, interfaces, and other module objects. |
| File name | somir.idl |
| Local-only | True |
| Ancestor interfaces | "Contained Interface" on page 79 "Container Interface" on page 85 |
| Exceptions | "CORBA::SystemException" on page |
| Supported operations | "ModuleDef::describe" on page 140 |

Intended Usage

The ModuleDef interface is used within the Interface Repository to represent an OMG IDL module. A ModuleDef object can be created using the create_module operation defined for the Container interface.

IDL syntax

```
module CORBA {
    interface ModuleDef:Container, Contained {
    };
    struct ModuleDescription {
        Identifier name;
        RepositoryId id;
        RepositoryId defined_in;
        VersionSpec version;
    };
};
```

ModuleDef::describe

| | |
|---------------------------|---|
| Overview | The describe operation returns a structure containing information about a CORBA::ModuleDef Interface Repository object. |
| Original interface | "ModuleDef Interface" on page 140 |
| Exceptions | "CORBA::SystemException" on page |

Intended Usage

The inherited describe operation returns a structure (CORBA::Contained::Description) that contains information about a CORBA::ModuleDef Interface Repository object. The CORBA::Contained::Description structure has two fields: kind (CORBA::DefinitionKind data type), and value (CORBA::Any data type).

The kind of definition described by the returned structure is provided using the kind field, and the value field is a CORBA::Any that contains the description that is specific to the kind of object described. When the describe operation is invoked on a module (CORBA::ModuleDef) object, the kind field is equal to CORBA::dk_Module and the value field contains the CORBA::ModuleDescription structure.

IDL Syntax

```

struct ModuleDescription {
    Identifier name;
    RepositoryId id;
    RepositoryId defined_in;
    VersionSpec version;
};
struct Description {
    DefinitionKind kind;
    any value;
};
Description describe ();

```

Input parameters

None.

Return values

Description *

The returned value is a pointer to a CORBA::Contained::Description structure. The memory is owned by the caller and can be removed using delete.

Example

```

// C++
// assume that 'this_module' has already been initialized
CORBA::ModuleDef * this_module;
// retrieve a description of the module
CORBA::Contained::Description * returned_description;
returned_description = this_module-> describe ();
// retrieve the module description from the returned description structure
CORBA::ModuleDescription * module_description;
module_description = (CORBA::ModuleDescription * ) returned_description
                    value.value ();

```

CORBA module: NamedValue Class

| | |
|--------------------------|--|
| Overview | Represents a request parameter, request return value, or Context property. |
| File name | nvlist.h |
| Supported methods | "NamedValue::_duplicate" on page 142 |
| | "NamedValue::_nil" on page 142 |
| | "NamedValue::flags" on page 142 |
| | "NamedValue::name" on page 143 |
| | "NamedValue::value" on page 143 |

Intended Usage

A Dynamic Invocation Interface request is comprised of an object reference, an operation, a list of arguments for the operation, and a return value. A NamedValue is used to represent each element of the argument list and the return value. A NamedValue is also used to represent each element of the property list associated with a Context. The ORB::create_named_value method is used to create an empty NamedValue. The NVList class provides methods to manage a list of named values. For additional information, see the NVList, Request, Context, and ORB class descriptions.

NamedValue::_duplicate

| | |
|----------------|---|
| Overview | Duplicates a NamedValue object. |
| Original class | "CORBA::NamedValue" on page 141 |

Intended Usage

This method is intended to be used by client and server applications to duplicate a reference to a NamedValue object. Both the original and the duplicate reference should subsequently be released using CORBA::release(NamedValue_ptr).

IDL Syntax

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

Input parameters

p

The NamedValue object to be duplicated. The reference can be nil, in which case the return value will also be nil.

Return values

CORBA::NamedValue_ptr

The new NamedValue object reference. This value should subsequently be released using CORBA::release(NamedValue_ptr).

NamedValue::_nil

| | |
|----------------|---|
| Overview | Returns a nil CORBA::NamedValue reference. |
| Original class | "CORBA::NamedValue" on page 141 |

Intended Usage

This method is intended to be used by client and server applications to create a nil NamedValue reference.

IDL Syntax

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

Input parameters

None.

Return values

CORBA::NamedValue_ptr

A nil NamedValue reference.

NamedValue::flags

| | |
|----------------|--|
| Overview | Returns a bitmask that identifies the argument passing mode. |
| Original class | "CORBA::NamedValue" on page 141 |

Intended Usage

The flags method is used by a client program when querying a NamedValue representing a parameter of a DII request. The flags method returns a bitmask that identifies the argument passing mode.

IDL Syntax

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

Input parameters

None.

Return values

CORBA::Flags

The argument passing mode. If the flags method is called on a NamedValue which does not represent a request parameter, an empty bitmask is returned. The following flag values are defined:

CORBA::ARG_IN

The associated value is an input only argument.

CORBA::ARG_OUT

The associated value is an output only argument.

CORBA::ARG_INOUT

The associated value is an in/out argument.

NamedValue::name

| | |
|----------------|---|
| Overview | Returns the argument name. |
| Original class | "CORBA::NamedValue" on page 141 |

Intended Usage

The name method is used by a client program when querying a NamedValue associated with a DII request. The name method returns the argument name, which is optional. The argument name in a NamedValue, if present, matches the argument name specified in the IDL definition of the operation.

IDL Syntax

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

Input parameters

None.

Return values

const char *

The argument name, if any, or a null pointer. Ownership of the return value is maintained by the NamedValue; the return value must not be freed by the caller.

NamedValue::value

| | |
|----------------|---|
| Overview | Returns the argument value. |
| Original class | "CORBA::NamedValue" on page 141 |

Intended Usage

The value method is used by a client program when querying a NamedValue associated with a DII request. The value method returns the argument value, which is accessed using standard operations on the Any class.

IDL Syntax

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

Input parameters

None.

Return values

CORBA::Any *

A pointer to the argument value, if any, or a null pointer. Ownership of the return value is maintained by the NamedValue; the return value must not be freed by the caller.

CORBA module: NVList Class

| | |
|--------------------------|--|
| Overview | Specifies a list of arguments: parameters associated with a request or properties associated with a Context. |
| File name | nvlist |
| Supported methods | "NVList::_duplicate" on page 144 |
| | "NVList::_nil" on page 145 |
| | "NVList::add" on page 145 |
| | "NVList::add_item" on page 146 |
| | "NVList::add_item_consume" on page 146 |
| | "NVList::add_value" on page 147 |
| | "NVList::add_value_consume" on page 148 |
| | "NVList::count" on page 148 |
| | "NVList::get_item_index" on page 149 |
| | "NVList::item" on page 149 |
| | "NVList::remove" on page 150 |

Intended Usage

A Dynamic Invocation Interface request is comprised of an object reference, an operation, a list of arguments for the operation, and a return value. An NVList is used to specify the list of arguments for the operation. An NVList is also used to specify the list of properties associated with a Context. The ORB::create_list method is called to create an empty named value list. The ORB::create_operation_list method is called to create a named value list for a specific operation. The NVList class provides methods to add and delete a named value, as well as query information about a named value list. For additional information, see the NamedValue, Request, Context, and ORB class descriptions.

NVList::_duplicate

| | |
|-----------------------|---|
| Overview | Duplicates an NVList object. |
| Original class | "CORBA::NVList" on page 144 |

Intended Usage

This method is intended to be used by client and server applications to duplicate a reference to an NVList object. Both the original and the duplicate reference should subsequently be released using CORBA::release(NVList_ptr).

IDL Syntax

```
static CORBA::NVList_ptr _duplicate (CORBA::NVList_ptr p);
```

Input parameters

p

The NVList object to be duplicated. The reference can be nil, in which case the return value will also be nil.

Return values

CORBA::NVList_ptr

The new NVList object reference. This value should subsequently be released using CORBA::release(NVList_ptr).

NVList::_nil

| | |
|----------------|---|
| Overview | Returns a nil CORBA::NVList reference. |
| Original class | "CORBA::NVList" on page 144 |

Intended Usage

This method is intended to be used by client and server applications to create a nil NVList reference.

IDL Syntax

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

Input parameters

None.

Return values

CORBA::NVList_ptr

A nil NVList reference.

NVList::add

| | |
|----------------|---|
| Overview | Adds an element to the end of a named value list. |
| Original class | "CORBA::NVList" on page 144 |

Intended Usage

The add method is used by a client program to populate the NVList associated with a DII request. The add method adds an element to the end of a named value list. The newly created named value is empty, except for the flags. See also the add_item, add_item_consume, add_value, and add_value_consume methods, which perform the same task but differ in memory management and how the newly created named value is initialized.

IDL Syntax

```
CORBA::NamedValue_ptr add(CORBA::Flags flags);
```

Input parameters

flags

A bitmask describing the argument. The following standard flag values identify the argument passing mode:

CORBA::ARG_IN

The associated value is an input-only argument.

CORBA::ARG_OUT

The associated value is an output-only argument.

CORBA::ARG_INOUT

The associated value is an in/out argument.

Return values

CORBA::NamedValue_ptr

A pointer to the newly created named value. Ownership of the return value is maintained by the NVList; the return value must not be freed by the caller.

NVList::add_item

| | |
|----------------|---|
| Overview | Adds an element to the end of a named value list. |
| Original class | "CORBA::NVList" on page 144 |

Intended Usage

The add_item method is used by a client program to populate the NVList associated with a DII request. The add_item method adds an element to the end of a named value list. The newly created named value is initialized using the input argument name and flags. The difference between the add_item and add_item_consume methods is that the former does not assume ownership of the input argument name, while the latter does. See also the add, add_value, and add_value_consume methods.

IDL Syntax

```
CORBA::NamedValue_ptr add_item(const char *id, CORBA::Flags flags);
```

Input parameters

flags

A bitmask describing the argument. The following standard flag values identify the argument passing mode:

CORBA::ARG_IN

The associated value is an input-only argument.

CORBA::ARG_OUT

The associated value is an output-only argument.

CORBA::ARG_INOUT

The associated value is an in/out argument.

Return values

CORBA::NamedValue_ptr

A pointer to the newly created named value. Ownership of the return value is maintained by the NVList; the return value must not be freed by the caller.

NVList::add_item_consume

| | |
|----------------|---|
| Overview | Adds an element to the end of a named value list. |
| Original class | "CORBA::NVList" on page 144 |

Intended Usage

The add_item_consume method is used by a client program to populate the NVList object associated with a DII request. The add_item_consume method adds an element to the end of a named value list. The newly created named value is initialized using the input argument name and flags. The difference between the add_item and add_item_consume methods is that the former does not assume ownership of the input argument name, while the latter does. The caller may not access the memory referred to by the input parameter after it has been passed in. See also the add, add_value, and add_value_consume methods.

IDL Syntax

```
CORBA::NamedValue_ptr add_item_consume(char *id, CORBA::Flags flags);
```

Input parameters

id

The name of the argument to be added. It is legal to pass a null pointer. If specified,

the input name should match the argument name specified in the IDL definition for the operation. The argument name must be allocated using the CORBA::string_alloc method. Ownership of this parameter transfers to the NVList.

flags

A bitmask describing the argument. The following standard flag values identify the argument passing mode:

CORBA::ARG_IN

The associated value is an input-only argument.

CORBA::ARG_OUT

The associated value is an output-only argument.

CORBA::ARG_INOUT

The associated value is an in/out argument.

Return values

CORBA::NamedValue_ptr

A pointer to the newly created named value. Ownership of the return value is maintained by the NVList; the return value must not be freed by the caller.

NVList::add_value

| | |
|----------------|---|
| Overview | Adds an element to the end of a named value list. |
| Original class | "CORBA::NVList" on page 144 |

Intended Usage

The add_value method is used by a client program to populate the NVList associated with a DII request. The add_value method adds an element to the end of a named value list. The newly created named value is initialized using the input argument name, value, and flags. The difference between the add_value and add_value_consume methods is that the former does not assume ownership of the input argument name and value, while the latter does.

See also ["NVList::add" on page 145](#) , ["NVList::add_item" on page 146](#) , and ["NVList::add_item_consume" on page 146](#) .

IDL Syntax

```
CORBA::NamedValue_ptr add_value(const char *id,
                               const CORBA::Any &any,
                               CORBA::Flags flags);
```

Input parameters

id

The name of the argument to be added. It is legal to pass a null pointer. If specified, the input name should match the argument name specified in the IDL definition for the operation.

any

The address of the value of the argument. It is legal to pass a null pointer.

flags

A bitmask describing the argument. The following standard flag values identify the argument passing mode:

CORBA::ARG_IN

The associated value is an input-only argument.

CORBA::ARG_OUT

The associated value is an output-only argument.

CORBA::ARG_INOUT

The associated value is an in/out argument.

Return values

CORBA::NamedValue_ptr

A pointer to the newly created named value. Ownership of the return value is maintained by the NVList; the return value must not be freed by the caller.

NVList::add_value_consume

| | |
|-----------------------|---|
| Overview | Adds an element to the end of a named value list. |
| Original class | "CORBA::NVList" on page 144 |

Intended Usage

The `add_value_consume` method is used by a client program to populate the NVList associated with a DII request. The `add_value_consume` method adds an element to the end of a named value list. The newly created named value is initialized using the input argument name, value, and flags. The difference between the `add_value` and `add_value_consume` methods is that the former does not assume ownership of the input argument name and value, while the latter does. The caller may not access the memory referred to by the input parameters after they have been passed in.

See also ["NVList::add" on page 145](#) , ["NVList::add_item" on page 146](#) , and ["NVList::add_item_consume" on page 146](#) .

IDL Syntax

```
CORBA::NamedValue_ptr add_value_consume(char *id,  
                                         CORBA::Any_ptr any,  
                                         CORBA::Flags flags);
```

Input parameters

id

The name of the argument to be added. It is legal to pass a null pointer. If specified, the input name should match the argument name specified in the IDL definition for the operation. Ownership of this parameter transfers to the NVList.

any

The address of the value of the argument. It is legal to pass a null pointer. Ownership of this parameter transfers to the NVList.

flags

A bitmask describing the argument. The following standard flag values identify the argument passing mode:

CORBA::ARG_IN

The associated value is an input-only argument.

CORBA::ARG_OUT

The associated value is an output-only argument.

CORBA::ARG_INOUT

The associated value is an in/out argument.

Return values

CORBA::NamedValue_ptr

A pointer to the newly created named value. Ownership of the return value is maintained by the NVList; the return value must not be freed by the caller.

NVList::count

| | |
|-----------------------|---|
| Overview | Returns the number of elements in a named value list. |
| Original class | "CORBA::NVList" on page 144 |

Intended Usage

The count method is used by a client program when querying the NVList associated with a DII request. The count method reports the number of elements in a named value list.

IDL Syntax

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

Input parameters

None.

Return values

CORBA::ULong

The number of elements in the named value list.

NVList::get_item_index

| | |
|-----------------------|--|
| Overview | Returns the index of the specified named value. |
| Original class | "CORBA::NVList" on page 144 |
| Exceptions | "CORBA::SystemException" on page |

Intended Usage

The `get_item_index` method is used by a client program when querying the NVList associated with a DII request. The `get_item_index` method returns the index of the specified named value. The argument name comparison is case insensitive. This method is an IBM extension to the CORBA 2.1 specification.

IDL Syntax

```
CORBA::Long get_item_index(const char *id);
```

Input parameters

id

The argument name of the desired named value. A system exception is raised if a null pointer is passed for this parameter.

Return values

CORBA::Long

The index of the specified named value. If the named value list is empty or the input argument name is not found, -1 is returned.

NVList::item

| | |
|-----------------------|--|
| Overview | Returns the named value associated with the input index. |
| Original class | "CORBA::NVList" on page 144 |
| Exceptions | "CORBA::SystemException" on page |

Intended Usage

The item method is used by a client program when querying the NVList associated with a DII request. The item method returns the named value associated with the input index.

IDL Syntax

```
CORBA::NamedValue_ptr item(CORBA::ULong index);
```

Input parameters

index

The index of the desired named value, starting at zero. A system exception is raised if the input index is greater than or equal to the number of elements in the named value list.

Return values

CORBA::NamedValue_ptr

A pointer to the named value associated with the input index. Ownership of the return value is maintained by the NamedValue; the return value must not be freed by the caller.

NVList::remove

| | |
|-----------------------|--|
| Overview | Deletes the named value associated with the input index. |
| Original class | "CORBA::NVList" on page 144 |
| Exceptions | "CORBA::SystemException" on page |

Intended Usage

The remove method is used by a client program when freeing an element of the NVList associated with a DII request. The remove method deletes the named value associated with the input index. CORBA::release(NamedValue_ptr) is called on the named value element. The remaining named value elements are re-indexed.

IDL Syntax

```
CORBA::Status remove(CORBA::Ulong index);
```

Input parameters

index

The index of the named value to be deleted, starting at zero. A system exception is raised if the input index is greater than or equal to the number of elements in the named value list.

Return values

CORBA::Status

A zero return code indicates the named value was successfully deleted.

CORBA module: Object Class

| | |
|--------------------------|---|
| Overview | Provides behavior common to all object references (both local objects and proxies to remote objects). |
| File name | object.h |
| Supported methods | "Object::_create_request" on page 151 |
| | "Object::_duplicate" on page 153 |
| | "Object::_get_implementation" on page 153 |
| | "Object::_get_interface" on page 154 |
| | "Object::_hash" on page 155 |
| | "Object::_is_a" on page 155 |
| | "Object::_is_equivalent" on page 156 |
| | "Object::_narrow" on page 157 |
| | "Object::_nil" on page 157 |
| | "Object::_non_existent" on page 158 |

["Object::_request" on page 158](#)

["Object::_this" on page 159](#)

Intended Usage

The CORBA::Object class is the abstract base class for all object references. This includes all proxy classes (objects, residing in client processes, that refer to remote objects residing in a server) and all classes that implement IDL interfaces to be exported from a server. As such, the CORBA::Object class provides methods that are meaningful to both local objects and (references to) remote objects.

Constants

```
static const char* Object_CN;
```

Object::_create_request

| | |
|----------------|---|
| Overview | Creates a Request object suitable for invoking a specific operation using the Dynamic Invocation Interface (DII). |
| Original class | "CORBA::Object" on page 150 |
| Exceptions | "CORBA::SystemException" on page |

Intended Usage

The `_create_request` method (two forms) are used to create CORBA::Request objects tailored to a specific IDL operation. The CORBA::Request object can then be used to invoke requests using the DII. After invoking the method, an application can obtain the return result, output parameter values, and exceptions using methods on CORBA::Request.

The target of CORBA::Object::_create_request() is typically a proxy object, rather than a local object. When invoked on a proxy object, this method operates locally; the remote object to which the proxy refers is unaffected until the DII request that is created by CORBA::Object::_create_request() is invoked.

The two forms of CORBA::Object::_create_request() differ in whether a CORBA::ExceptionList_ptr and a CORBA::ContextList_ptr are provided as input. These input parameters are not needed for operations that have no `raises` or `context` clause in the IDL specification. For IDL operations that do have a `raises` or `context` clause, the second form of CORBA::Object::_create_request() can be used to avoid (potentially time-consuming) Interface Repository lookups by the ORB when the DII request is invoked.

See also ["Object::_request" on page 158](#) which creates a CORBA::Request without providing the parameters for the operation.

IDL Syntax

```
virtual CORBA::Status _create_request (CORBA::Context_ptr ctx,
                                       const char* operation,
                                       CORBA::NVList_ptr arg_list,
                                       CORBA::NamedValue_ptr result,
                                       CORBA::Request_ptr &request,
                                       CORBA::Flags reg_flags) = 0;

virtual CORBA::Status _create_request (CORBA::Context_ptr ctx,
                                       const char* operation,
                                       CORBA::NVList_ptr arg_list,
                                       CORBA::NamedValue_ptr result,
                                       CORBA::ExceptionList_ptr exc_list,
                                       CORBA::ContextList_ptr ctx_list,
                                       CORBA::Request_ptr &request,
                                       CORBA::Flags reg_flags) = 0;
```

Input parameters

ctx

A pointer to the CORBA::Context object to be passed when the DII request is invoked. For operations having no `context` clause in their IDL specification, this can be NULL. The CORBA::Request object assumes ownership of this parameter.

operation

The unscoped name of the IDL operation to be invoked using the Request. This must be an operation that is implemented or inherited by the CORBA::Object on which CORBA::Object::_create_request() is invoked. The caller retains ownership of this parameter (the CORBA::Request object makes a copy).

arg_list

A pointer to a CORBA::NVList object that describes;

- The types of all the operation parameters of the IDL.
- The values of the *in* and *inout* parameters of the operation.
- The variables in which the *out* parameter values will be stored after the DII request is invoked.

result

A pointer to a CORBA::NamedValue object that will hold the result of the DII request after it is invoked. The CORBA::Request object assumes ownership of this parameter.

request

A CORBA::Request_ptr variable, passed by reference, to be initialized by CORBA::Object::_create_request() to point to the newly-created CORBA::Request object. The caller retains ownership of this parameter.

req_flags

A bit-vector describing how the DII request will be invoked. Oneway methods (methods that do not require a response) should be created using a *req_flags* value of CORBA::INV_NO_RESPONSE. No other *req_flags* values are currently used.

exc_list

A pointer to a CORBA::ExceptionList object that describes the user-defined exceptions that the DII operation can throw (according to the operation declaration in the IDL specification). This parameter is essentially a list of TypeCodes for UserException subclasses. The CORBA::Request object assumes ownership of this parameter. This parameter is optional and NULL can be passed (even for methods that raise user-defined exceptions).

ctx_list

A pointer to a CORBA::ContextList object that lists the Context strings that must be sent with the DII operation (according to the operation declarations in the IDL specification). This parameter differs from the *ctx* parameter in that this parameter supplies only the context strings whose values are to be transmitted with the DII request, while the *ctx* parameter is the object from which those context string values are obtained. The CORBA::Request object assumes ownership of this parameter. This parameter is optional and NULL can be passed (even for methods that pass Context parameters).

Return values

CORBA::Status

A return value of zero indicates success.

Example

```
/* The following IDL signature is used:
   interface testObject
   {
       string testMethod(in long input_value, out float outvalue);
   };
*/
...
/* Get the OperationDef that describes testMethod */
CORBA::ORB_var myorb =
    CORBA::ORB_init(argc, argv, "DSOM"); /* argc, argv: input arguments */
CORBA::Repository_var my_IR = CORBA::Repository::_narrow(generic_IR);
CORBA::Contained_var generic_opdef =
    my_IR -> lookup("testObject::testMethod");
CORBA::OperationDef_var my_opdef =
    CORBA::OperationDef::_narrow(generic_opdef);
```

```

/* Create the NVList and NamedValue for the request */
CORBA::NVList_ptr params = NULL;
myorb -> create_operation_list(my_opdef, params);
CORBA::NamedValue_ptr result = NULL;
myorb -> create_named_value(result);
/* Create the Request object */
CORBA::Object_var my_proxy = /* Get a proxy somehow */
my_proxy -> _create_request(NULL, "testMethod", params, result,
my_request, 0);

```

Object::_duplicate

| | |
|-----------------------|---|
| Overview | Duplicates an object reference. |
| Original class | "CORBA::Object" on page 150 |

Intended Usage

This method is intended to be used by client and server applications, to duplicate object references (both pointers to local implementation objects and proxies to remote objects). For each duplication performed on an object reference, an equal number of calls to CORBA::release must also be made for the reference to be deleted.

When an application passes an object reference (either a local object or a proxy) on a method call, either as a parameter value or a return result, if the call transfers ownership of the object reference and the application needs to retain ownership of the reference as well, the application should first duplicate the reference before passing it. Each user of the reference should subsequently CORBA::release the reference so that its resources can be reclaimed.

When CORBA::Object::_duplicate is called on a proxy object, only the proxy is affected; no remote invocation is made to the remote object to which the proxy refers. Hence, CORBA::Object::_duplicate and CORBA::release are only used to manage the local resources associated with object references.

IDL Syntax

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

Input parameters

obj

The object reference to be duplicated., If this parameter is a nil object reference (NULL), no action is taken.

Return values

CORBA::Object_ptr

The duplicate of the input object reference. (Because CORBA::Object::_duplicate and CORBA::release are implemented using reference counting, this will be the same as the input value.) If the input value is a nil reference, the return value will likewise be nil.

Example

```

/*The following example is written in C++*/
#include "corba.h"
/* this function returns duplicate of an object ref */
CORBA::Object_ptr getObj(CORBA::Object_ptr p)
{
    return CORBA::Object::_duplicate(p);
}

```

Object::_get_implementation

| | |
|-----------------------|---|
| Overview | Returns a reference to the CORBA::ImplementationDef describing the server in which a remote object resides. |
| Original class | "CORBA::Object" on page 150 |
| Exceptions | "CORBA::SystemException" on page |

Intended Usage

This method is intended to be used to obtain the CORBA::ImplementationDef object

describing the server in which a remote object resides. When invoked on a proxy object, this method is forwarded to the remote object, and a proxy to a remote CORBA::ImplementationDef object (residing in the same server as the remote object) is returned. When invoked on a local object residing in a server, the local CORBA::ImplementationDef object (the one originally passed to CORBA::BOA::impl_is_ready) is returned. When invoked on a local object in a client (that is not also a server), NULL is returned.

IDL Syntax

```
virtual CORBA::ImplementationDef_ptr _get_implementation () = 0;
```

Input parameters

None.

Return values

CORBA::ImplementationDef_ptr

A pointer to the ImplementationDef object that describes the server in which an object (referred to by an object reference) resides. The caller assumes ownership of this object, and should subsequently CORBA::release (not delete) it.

Example

```
/* The following is a C++ example */
#include "corba.h"
#include <string.h>
/* Assume p is a proxy object pointer derived from CORBA::Object class
   the following will get the impl def and interface def on remote objects
   */
CORBA::ImplementationDef_ptr impl;
CORBA::InterfaceDef_ptr intf;
string str;
impl = p->_get_implementation();
if(impl)
{
    str = impl->get_alias();          /* get implementation alias */
    /* ensure it's the right impl and work with the impl */
    ...
}
else /* generate exception */ ...
    intf = p->_get_interface();
    if(intf)
    {
        str = intf->id();            /* get interface id */
        /* ensure it's the right interface and work with the intf */
        ...
    }
else /* generate exception */ ...
CORBA::release(p);
```

Object::_get_interface

| | |
|-----------------------|---|
| Overview | Returns a reference to the CORBA::InterfaceDef describing the most specific interface supported by the target object. |
| Original class | "CORBA::Object" on page 150 |
| Exceptions | "CORBA::SystemException" on page |

Intended Usage

This method is intended to be used to obtain the CORBA::InterfaceDef object describing the most specific interface of the target object. When invoked on a proxy object, this method is forwarded to the remote object, and a proxy to a remote CORBA::InterfaceDef object (residing in the same server as the remote object) is returned. This InterfaceDef describes the interface of the remote object, which may be more specific than the interface of the proxy on which CORBA::_get_interface was invoked. (This can occur when the client does not have bindings for the most specific interface supported by the remote object.)

When invoked on a local object, a local CORBA::InterfaceDef object is retrieved from the local Interface Repository.

IDL Syntax

```
virtual CORBA::InterfaceDef_ptr _get_interface () = 0;
```

Input parameters

None.

Return values

CORBA::InterfaceDef_ptr

A pointer to the InterfaceDef object that describes the most specific interface supported by the target object . The caller assumes ownership of this object.

Example

See the CORBA::“ [Object::_get_implementation](#)” on page 153 .

Object::_hash

| | |
|----------------|---|
| Overview | Maps object references into disjoint groups of potentially equivalent references. |
| Original class | “CORBA::Object” on page 150 |

Intended Usage

This method is intended to be used by applications that manipulate large numbers of object references, for mapping object references into disjoint groups of potentially equivalent references. The hash value of an object reference does not change during the lifetime of the reference. The hash value of an object reference is not necessarily unique (another reference may have the same hash value). Different object references to the same remote object do not necessarily hash to the same value.

When invoked on a proxy object, this method does not result in a remote request to the server; all processing is done locally.

IDL Syntax

```
virtual CORBA::ULong _hash (CORBA::ULong maximum) = 0;
```

Input parameters

maximum

The upper bound on the return value.

Return values

CORBA::ULong

A hash value with a lower bound of zero and an upper bound as indicated by the maximum parameter.

Example

```
/*The following example is written in C++*/  
#include "corba.h"  
#define HASH_MAX 10000  
/* assume p is CORBA::Object pointer */  
::CORBA::ULong hash_ulong = 0;  
hash_ulong = p->_hash(HASH_MAX);  
...
```

Object::_is_a

| | |
|----------------|--|
| Overview | Determines whether an object supports a given IDL interface. |
| Original class | “CORBA::Object” on page 150 |
| Exceptions | “CORBA::SystemException” on page |

Intended Usage

This method is intended to be used by applications to determine whether an object reference refers to an object that supports a given IDL interface (and hence whether the

reference can be successfully narrowed). When invoked on a proxy object, this call sometimes results in a remote invocation (if it cannot be determined locally).

IDL Syntax

```
virtual CORBA::Boolean _is_a (const char* logical_type_id) = 0;
```

Input parameters

logical_type_id

The Interface Repository type identifier of an IDL interface. This is not simply the interface name. For programmer convenience, type identifiers are provided by the C++ bindings, as static consts of the C++ class corresponding to the interface, using the naming convention <interface-name>::<interface-name>_CN. If this parameter value is NULL or not a valid Interface Repository type identifier, zero is returned.

Return values

CORBA::Boolean

A zero return value indicates that the object referenced by the object reference does not support the specified IDL interface. A nonzero return value indicates that it does support it. An object is considered to support the interface if it either implements it or inherits it.

Example

```
/* Assume the following idl interface: */
interface testObject {
    string testMethod (in long input_value, out float out_value);
};
/* Here is the cpp code: */
CORBA::Object_ptr test_obj;
/* initialize test_obj somehow */

/** To find out if test_obj can be narrowed to testObject, use
    CORBA::Object::_is_a and the Repository ID for the testObject interface
    (defined in the emitted bindings as testObject::testObject_RID) */
if (test_obj->_is_a(testObject::testObject_RID))
    testObject_ptr new_test_obj = testObject::_narrow(test_obj);
...
```

Object::_is_equivalent

| | |
|-----------------------|--|
| Overview | Determines whether two object references refer to the same object. |
| Original class | "CORBA::Object" on page 150 |

Intended Usage

This method is intended to be used by applications to determine whether two object references refer to the same object, as far as the ORB can easily determine. In the case of proxies, this method attempts to determine whether two proxies refer to the same remote object. When invoked on proxy objects, this method operates locally and does not involve the remote object to which the proxy refers. For this reason, it is possible for this method to return zero, indicating that the two references do not appear to be equivalent, when in fact they are equivalent (but it cannot be determined without communicating with the remote server).

IDL Syntax

```
virtual CORBA::Boolean _is_equivalent (const
    CORBA::Object_ptr other_object) = 0;
```

Input parameters

other_object

An object reference to be compared to the target object reference.

Return values

CORBA::Boolean

A zero return value indicates that the target object reference does not refer to the same object as the given object reference, as far as the ORB can easily determine. (It

is still possible that the two object references are equivalent, however.) A non-zero return value indicates that the target object reference and the given object reference do refer to the same object.

Example

```

/*The following example is written in C++*/
#include "corba.h"
CORBA::Object_ptr p1;
CORBA::Object_ptr p2;
/*construct p1 and p2 */
...
/* check to see if they are different objects */
CORBA::Boolean retval = p1->_is_equivalent(p2);

```

Object::_narrow

| | |
|-----------------------|--|
| Overview | Performs essentially the same function as CORBA::Object::_duplicate(). |
| Original class | "CORBA::Object" on page 150 |
| Exceptions | "CORBA::SystemException" on page |

Intended Usage

This method is provided for consistency with the _narrow methods provided by the C++ bindings for subclasses of CORBA::Object, which narrow a generic CORBA::Object to a more specific type. When narrowing from a CORBA::Object to a CORBA::Object, however, the method degenerates to a simple duplication. Hence, this method is equivalent to CORBA::Object::_duplicate.

IDL Syntax

```

static CORBA::Object_ptr _narrow (CORBA::Object_ptr obj);

```

Input parameters

obj

The CORBA::Object to be narrowed. The caller retains ownership of this object reference.

Return values

CORBA::Object_ptr

The narrowed (and duplicated) object reference. The caller assumes ownership of this object reference and should subsequently CORBA::release it.

Example

```

/* Assume the following idl interface: */
interface testObject
{
    string testMethod (in long input_value, out float out_value);
};
/* Here is the cpp code: */
CORBA::Object_ptr optr;
/* instantiate optr somehow */
...
testObject_ptr test_obj = testObject::_narrow(optr);
...

```

Object::_nil

| | |
|-----------------------|---|
| Overview | Returns a nil CORBA::Object reference. |
| Original class | "CORBA::Object" on page 150 |

Intended Usage

This method is intended to be used by client and server applications to create a nil Object reference. Since a nil value proxy object may be generated and returned by this call (versus a NULL), nil references can and should be released when no longer required by the client application. Due to this "variable" returned value, client and server applications should be using the CORBA::is_nil() method for checking for nil references (instead of checking against NULL).

IDL Syntax

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

Input parameters

None.

Return values

CORBA::Object_ptr

A nil Object reference.

Example

```
/* Assume the following IDL interface */  
interface testObject  
{  
    string testMethod ( in long input_value, out float out_value);  
};  
/* Here is the cpp code */  
testObject_ptr test_obj = testObject::_nil();  
...
```

Object::_non_existent

| | |
|-----------------------|--|
| Overview | Determines whether an object reference refers to a valid object. |
| Original class | "CORBA::Object" on page 150 |

Intended Usage

This method is intended to be used to determine whether an object reference (either a proxy object or a local pointer) refers to a valid object. When invoked on a proxy object, this results in a remote call to the server, which may activate the remote object to determine its existence, but no method is invoked on the remote object itself (unless the server invokes some method on the object as part of activation).

IDL Syntax

```
virtual Boolean _non_existent () = 0;
```

Input parameters

None.

Return values

CORBA::Boolean

A zero return value indicates that the target object reference refers to a valid object. A nonzero return value indicates that the target object reference refers to a non-existent object.

Example

```
/* Assume the following IDL interface */  
interface testObject  
{  
    string testMethod (in long input_value, out float out_value);  
};  
/* Here is the cpp code */  
testObject_ptr test_obj;  
/* instantiate test_obj and make it a proxy somehow */  
...  
CORBA::Boolean retval = test_obj->_non_existent();  
...
```

Object::_request

| | |
|-----------------------|---|
| Overview | Creates a Request object suitable for invoking a specific operation using the Dynamic Invocation Interface (DII). |
| Original class | "CORBA::Object" on page 150 |
| Exceptions | "CORBA::SystemException" on page |

Intended Usage

The `_request` method is used to create a `CORBA::Request` object tailored to a specific IDL operation. Arguments, the operation's return type, and context identifiers should be added after construction via methods on `CORBA::Request`. The `CORBA::Request` object can then be used to invoke requests using the Dynamic Invocation Interface (DII). After invoking the method, an application can obtain the return results, output parameter values, and exceptions via methods on `CORBA::Request`.

The target of the `CORBA::Object::_request` method is typically a proxy object, rather than a local object. When invoked on a proxy object, this method operates locally; the remote object to which the proxy refers is unaffected until the DII Request that is created by `CORBA::Object::_request` is invoked.

This mechanism for creating a `CORBA::Request` assumes that the operation to be invoked via the DII is not "oneway" (that a response is required).

See also [CORBA::"Object::_create_request" on page 151](#), which allows the `CORBA::Request` to be created and fully initialized at once.

IDL Syntax

```
virtual CORBA::Request_ptr _request (const char* operation) = 0;
```

Input parameters

operation

The unscoped name of the IDL operation to be invoked using the Request. This must be an operation that is implemented or inherited by the `CORBA::Object` on which `CORBA::Object::_create_request` is invoked. The caller retains ownership of this parameter (the `CORBA::Request` object makes a copy).

Return values

`CORBA::Request_ptr`

A pointer to the newly-created `CORBA::Request` object. The caller assumes ownership of this object and should subsequently delete it.

Example

```
/* The following C++ fragment creates a request object assuming
   that p is a proxy object pointer already declared and defined
   */
#include "corba.h"
CORBA::Request_ptr req = p->_request("dii_string_tst");
CORBA::String str = CORBA::string_alloc(12+1);
strcpy( str, "Input String");
req->add_in_arg() <<= str;
req->set_return_type(CORBA::_tc_string);
req->invoke();
...
CORBA::string_free(str);
CORBA::release(req);
```

Object::_this

| | |
|----------------|--|
| Overview | Returns a duplicated object reference for the object implementation on which this operation was invoked. |
| Original class | "CORBA::Object" on page 150 |

Intended Usage

This method is intended to be used within an implementation of an IDL interface, to obtain a duplicate of the object on which an operation was invoked. Calling `_this()` within an IDL operation implementation is not equivalent to calling `_duplicate(this)`, because an object reference is not necessarily the object itself. To be CORBA compliant, an implementation should use `_this()` instead of `_duplicate(this)`. In addition, even though `_this()` is implemented today as a non-virtual method on `CORBA::Object`, and on all the C++ interface classes generated for each interface, an implementation may not assume that it will always be implemented in this way. It may only assume that "`_this()`" is available within the scope of

the implementation, and will always return the correct object reference for the interface that corresponds to the implementation.

`_this()` may not be used by a client. A client who already holds an object reference may use `'InterfaceName'::_narrow(objref)` to obtain an object reference to a more derived interface whose name is `'InterfaceName'`. It can also rely on automatic C++ conversion to obtain an object reference to a parent interface.

IDL Syntax

```
CORBA::Object_ptr _this();
```

Input parameters

None.

Return values

CORBA::Object_ptr

A duplicate of the object reference on which `CORBA::Object::_this` was invoked. The caller assumes ownership of this object reference and should subsequently either `CORBA::release` it or transfer ownership of it to another party.

Example

```
/* Assume the following IDL interface */
interface testObject
{
    testObject testMethod ( );
};
/* Here is the cpp code that might appear in
   an implementation of testObject::testMethod
   */
CORBA::Object_ptr MyImplementation::testMethod()
{
    return _this();      /* duplicates and returns self */
}
...
```

CORBA module: OperationDef Interface

| | |
|-----------------------------|---|
| Overview | An OperationDef is used within the Interface Repository to represent the information needed to define an operation of an interface. |
| File name | somir.idl |
| Local-only | True |
| Ancestor interfaces | "Contained Interface" on page 79 |
| Exceptions | "CORBA::SystemException" on page |
| Supported operations | "OperationDef::contexts" on page 161 |
| | "OperationDef::describe" on page 162 |
| | "OperationDef::exceptions" on page 163 |
| | "OperationDef::mode" on page 163 |
| | "OperationDef::params" on page 164 |
| | "OperationDef::result" on page 165 |
| | "OperationDef::result_def" on page 166 |

Intended Usage

The OperationDef object is used to represent the information that defines an operation of an interface. An OperationDef may be created by calling the `create_operation` operation of the InterfaceDef interface . The `create_operation` parameters include the unique RepositoryId

(CORBA::RepositoryId), the name (CORBA::Identifier), the version (CORBA::VersionSpec), the result (CORBA::IDLType*) to indicate the type of the returned operation result, the mode of the operation (CORBA::OP_NORMAL or CORBA::OP_ONEWAY), a sequence (CORBA::ParDescriptionSeq) defining the parameters of the operation, a sequence (CORBA::ExceptionDefSeq) defining the exceptions of the operation, and a sequence (CORBA::ContextIdSeq) defining the contexts of the operation.

IDL syntax

```

module CORBA {
    enum OperationMode {OP_NORMAL, OP_ONEWAY};
    enum ParameterMode {PARAM_IN, PARAM_OUT, PARAM_INOUT};
    struct Parameter Description {
        Identifier name;
        TypeCode type;
        IDLType type_def;
        ParamterMode mode;
    };
    typedef sequence ParDescriptionSeq;
    typedef identifier ContextIdentifier;
    typedef sequence contextIdSeq;
    typedef sequence ExceptionDefSeq;
    typedef sequence ExcDescriptionSeq;
    interface OperationDef:Contained {
        readonlyattribute TypeCode result;
        attribute IDLType result_def;
        attribute ParDescriptionSeq params;
        attribute OperationMode mode;
        attribute ContextIdSeq contexts;
        attribute ExceptionDefSeq exceptions;
    };
    struct OperationDescription {
        Identifier name;
        RepositoryId id;
        RepositoryId defined_in;
        VersionSpec version;
        TypeCode result;
        OperationMode mode;
        ContextIdSeq contexts;
        ParDescriptionSeq parameters;
        ExcDescriptionSeq exceptions;
    };
};

```

OperationDef::contexts

| | |
|---------------------------|---|
| Overview | The context read and write operations allow the access and update of the list of context identifiers that apply to an operation (CORBA::OperationDef object) in the Interface Repository. |
| Original interface | " OperationDef Interface" on page 160 |
| Exceptions | "CORBA::SystemException" on page |

Intended Usage

The context attribute specifies the list of context identifiers that apply to an operation definition. The context read and write operations are supported with parameter and result definitions as described below.

IDL Syntax

```

attribute ContextIdSeq contexts;

```

Read operations

Input parameters

None.

Return values

CORBA::ContextIdSeq *

The returned value is a pointer to a copy of the contexts attribute of the operation definition. The memory is owned by the caller and can be removed by invoking delete.

Write operations

Input parameters

ContextIdSeq & contexts

The contexts parameter is the new list of contexts with which to update the operation definition (the length of the sequence may be set to zero to indicate no contexts).

Return values

None.

Example

```
// C++
// assume that 'this_operation' has already been initialized
CORBA::OperationDef * this_operation;
// establish the sequence of contexts for updating the operation
// definition
CORBA::ContextIdSeq seq_update;
seq_update.length (2);
seq_update[0] = CORBA::string_dup ("CONTEXT_0=value_0");
seq_update[1] = CORBA::string_dup ("CONTEXT_1= value_1");
// update the operation with the new contexts list
this_operation-> contexts (seq_update);
// retrieve the contexts list from the operation definition
CORBA::ContextIdSeq * returned_context_list;
returned_context_list = this_operation-> contexts ();
```

OperationDef::describe

| | |
|---------------------------|--|
| Overview | The describe operation returns a structure containing information about a CORBA::OperationDef Interface Repository object. |
| Original interface | "OperationDef Interface" on page 160 |
| Exceptions | "CORBA::SystemException" on page |

Intended Usage

The inherited describe operation returns a structure (CORBA::Contained::Description) that contains information about a CORBA::OperationDef Interface Repository object. The CORBA::Contained::Description structure has two fields: kind (CORBA::DefinitionKind data type), and value (CORBA::Any data type).

The kind of definition described by the returned structure is provided using the kind field, and the value field is a CORBA::Any that contains the description that is specific to the kind of object described. When the describe operation is invoked on an operation (CORBA::OperationDef) object, the kind field is equal to CORBA::dk_Operation and the value field contains the CORBA::OperationDescription structure.

IDL Syntax

```
struct OperationDescription {
    Identifier name;
    RepositoryId id;
    RepositoryId defined_in;
    VersionSpec version;
    TypeCode result;
    OperationMode mode;
    ContextIdSeq contexts;
    ParDescriptionSeq parameters;
    ExcDescriptionSeq exceptions;
};
struct Description {
    DefinitionKind kind;
    any value;
};
Description describe ();
```

Input parameters

None.

Return values

Description *

The returned value is a pointer to a CORBA::Contained::Description structure. The memory is owned by the caller and can be removed by invoking delete.

Example

```
// C++
```

```

// assume that 'this_operation' has already been initialized
CORBA::OperationDef * this_operation;
// retrieve a description of the operation
CORBA::OperationDef::Description * returned_description;
returned_description = this_operation-> describe ();
// retrieve the operation description from the returned description
// structure
CORBA::OperationDescription * operation_description;
operation_description = (CORBA::OperationDescription *)
    returned_description value.value ();

```

OperationDef::exceptions

| | |
|---------------------------|---|
| Overview | The exceptions read and write operations allow access and update of the list of exceptions associated with an operation definition (CORBA::OperationDef) within the Interface Repository. |
| Original interface | "OperationDef Interface" on page 160 |
| Exceptions | "CORBA::SystemException" on page |

Intended Usage

The exceptions attribute specifies the list of exception types that can be raised by the operation. The exceptions read and write operations are supported with parameter and result descriptions as defined below.

IDL Syntax

```
attribute ExceptionDefSeq exceptions;
```

Read operations

Input parameters

None.

Return values

CORBA::ExceptionDefSeq *

The returned value is a pointer to a copy of the exceptions attribute of the operation definition. The memory is owned by the caller and can be removed by invoking delete.

Write operations

Input parameters

CORBA::ExceptionDefSeq & exceptions

The exceptions parameter is the sequence of exceptions with which to update the exceptions attribute of the operation definition (the sequence length may be set to zero to indicate no exceptions for the operation).

Return values

None.

Example

```

// C++
// assume that 'this_operation' and 'this_exception'
// have already been defined
CORBA::OperationDef * this_operation;
CORBA::ExceptionDef * this_exception;
// establish the exception definition sequence to update the operation
CORBA::ExceptionDefSeq new_exceptions;
new_exceptions.length (1);
new_exceptions[0] = this_exception;
this_operation-> exceptions (new_exceptions);
// retrieve the exception list from the operation
CORBA::ExceptionDefSeq * returned_exception_list;
returned_exception_list = this_operation-> exceptions ();

```

OperationDef::mode

| | |
|-----------------|--|
| Overview | Access and update the mode attribute of an operation definition (CORBA::OperationDef) within the Interface |
|-----------------|--|

| | |
|---------------------------|--|
| | Repository. |
| Original interface | "OperationDef Interface" on page 160 |
| Exceptions | "CORBA::SystemException" on page |

Intended Usage

The mode read and write operations allow the access and update of the mode attribute of an operation definition (CORBA::OperationDef) within the Interface Repository.

The operation's mode attribute can be one of two values. If no output is returned by the operation, the operation is oneway (the mode attribute is CORBA::OP_ONEWAY), otherwise the operation is normal (the mode attribute is CORBA::OP_NORMAL).

The mode attribute can only be set to CORBA::OP_ONEWAY if the result is void and all of the operation parameters (the params attribute) are input only (have a mode of CORBA::PARAM_IN).

IDL Syntax

```
OperationMode mode;
```

Read operations

Input parameters

None.

Return values

CORBA::OperationMode

The returned value is the current value of the mode attribute of the operation definition (CORBA::OperationDef) object.

Write operations

Input parameters

mode

The mode parameter is the new value to which the mode attribute of the CORBA::OperationDef object will be set. Valid mode values include CORBA::OP_ONEWAY and CORBA::OP_NORMAL.

Return values

None.

Example

```
// C++
// assume that 'this_operation' has already been initialized
CORBA::OperationDef * this_operation;
// set the new mode in the operation definition
CORBA::OperationMode new_mode = CORBA::OP_NORMAL;
this_operation-> mode (new_mode);
// retrieve the mode from the operation definition
CORBA::OperationMode returned_mode;
returned_mode = this_operation-> mode ();
```

OperationDef::params

| | |
|---------------------------|---|
| Overview | The params read and write operations allow the access and update of the parameter descriptions of an operation definition object (CORBA::OperationDef) in the Interface Repository. |
| Original interface | "OperationDef Interface" on page 160 |
| Exceptions | "CORBA::SystemException" on page |

Intended Usage

The params attribute describes the parameters of the operation.

The params attribute is a CORBA::ParDescriptionSeq data type, each element of which has 4 fields. The name (CORBA::Identifier) is the name of the parameter. The type field references a CORBA::TypeCode that represents the parameter type. The type_def field references a CORBA::IDLType that represents the parameter type definition. The mode field defines the parameter as an input parameter, an output parameter, or as used for both input and output (CORBA::PARAM_IN, CORBA::PARAM_OUT, and CORBA::PARAM_INOUT, respectively). The order of the elements in the sequence is important and should reflect the actual order of the parameters in the operation signature.

The params read and write operations are supported with the parameters and return values as defined below.

IDL Syntax

```
attribute ParDescriptionSeq params;
```

Read operations

Input parameters

None.

Return values

CORBA::ParDescriptionSeq*

The returned sequence of CORBA::ParameterDescriptions is a copy of the params attribute of the CORBA::OperationDef object. The memory is owned by the caller and can be removed by calling delete.

Write operations

Input parameters

CORBA::ParDescriptionSeq & params

The params parameter defines the new list of parameters that will comprise the parameters for the operation.

Return values

None.

Example

```
// C++
// assume that 'this_operation' and 'pk_long_ptr'
// have already been initialized
CORBA::OperationDef * this_operation;
CORBA::PrimitiveDef * pk_long_ptr;
// establish the CORBA::ParDescriptionSeq
CORBA::ParDescriptionSeq seq_update;
seq_update.length (1);
seq_update[0].name = CORBA::string_dup ("parameter_0");
seq_update[0].type = CORBA::tc_long;
seq_update[0].type_def = CORBA::IDLType::_duplicate (pk_long_ptr);
seq_update[0].mode = CORBA::PARAM_IN;
// update the params attribute in the OperationDef
this_operation-> params (seq_update);
// retrieve the params attribute from the OperationDef
CORBA::ParDescriptionSeq * returned_parm_list;
returned_parm_list = this_operation-> params ();
```

OperationDef::result

| | |
|--------------------|--|
| Overview | The result read operation returns a type (CORBA::TypeCode *) representative of the value returned by the operation |
| Original interface | "OperationDef Interface" on page 160 |
| Exceptions | "CORBA::SystemException" on page |

Intended Usage

The result attribute of a CORBA::OperationDef object references a CORBA::TypeCode * that describes the type of the value returned by the operation. The result read operation can be used to retrieve a pointer to a copy of the CORBA::TypeCode referenced by the result attribute.

IDL Syntax

```
readonly attribute TypeCode result;
```

Input parameters

None.

Return values

TypeCode *

The returned value is a pointer to a copy of the CORBA::TypeCode referenced by the result attribute. The memory is owned by the caller and can be returned using CORBA::release.

Example

```
// C++
// assume that 'this_operation' has already been initialized
CORBA::OperationDef * this_operation;
// retrieve the TypeCode which represents the type of result
// of the operation
CORBA::TypeCode * operations_result_tc;
operations_result_tc = this_operation-> result ();
```

OperationDef::result_def

| | |
|---------------------------|--|
| Overview | The result_def read and write operations allow the access and update of the result type definition of an operation definition (CORBA::OperationDef) in the Interface Repository. |
| Original interface | "OperationDef Interface" on page 160 |
| Exceptions | "CORBA::SystemException" on page |

Intended Usage

The type of the result of an operation definition is identified by the result_def attribute (a reference to a CORBA::IDLType *). Read and write result_def operations are supported, the parameter and return value definitions of which are defined below.

IDL Syntax

```
attribute IDL/Type result_def;
```

Read operations

Input parameters

None.

Return values

CORBA::IDLType *

The returned object is a pointer to a copy of the CORBA::IDLType referenced by the result_def attribute of the CORBA::OperationDef object. The returned object is owned by the caller and can be released by invoking CORBA::release.

Write operations

Input parameters

CORBA::IDLType * result_def

The result_def parameter represents the new result definition for the CORBA::OperationDef. Setting the result_def attribute also updates the result attribute.

Return values

None.

Example

```
// C++
// assume that 'this_operation' and 'this_struct' have already been
// initialized
CORBA::OperationDef * this_operation;
CORBA::StructDef * this_struct;
// change the operation result type definition to 'this_struct'
this_operation-> result_def (this_struct);
// read the operation's result type definition from 'this_operation'
CORBA::IDLType * returned_result_def;
returned_result_def = this_operation-> result_def ();
```

CORBA module: ORB Class

| | |
|--------------------------|--|
| Overview | Provides basic Object Request Broker services. |
| File name | orb.h |
| Nested classes | RequestSeq |
| Supported methods | "ORB::_duplicate" on page 168 |
| | "ORB::_nil" on page 168 |
| | "ORB::BOA_init" on page 169 |
| | "ORB::create_alias_tc" on page 170 |
| | "ORB::create_array_tc" on page 171 |
| | "ORB::create_context_list" on page 171 |
| | "ORB::create_enum_tc" on page 172 |
| | "ORB::create_environment" on page 173 |
| | "ORB::create_exception_list" on page 173 |
| | "ORB::create_exception_tc" on page 174 |
| | "ORB::create_interface_tc" on page 175 |
| | "ORB::create_list" on page 175 |
| | "ORB::create_named_value" on page 176 |
| | "ORB::create_operation_list" on page 177 |
| | "ORB::create_recursive_sequence_tc" on page 177 |
| | "ORB::create_sequence_tc" on page 178 |
| | "ORB::create_string_tc" on page 179 |
| | "ORB::create_struct_tc" on page 180 |
| | "ORB::create_union_tc" on page 180 |
| | "ORB::get_default_context" on page 181 |
| | "ORB::get_next_response" on page 182 |
| | "ORB::get_service_information" on page 183 |
| | "ORB::list_initial_services" on page 184 |
| | "ORB::object_to_string" on page 184 |
| | "ORB::poll_next_response" on page 185 |
| | "ORB::resolve_initial_references" on page 186 |
| | "ORB::resolve_initial_references_remote" on page 186 |
| | "ORB::send_multiple_requests_deferred" on page 188 |

["ORB::send_multiple_requests_oneway" on page 188](#)

["ORB::string_to_object" on page 189](#)

Intended Usage

The ORB class is intended to be used by client and server applications to access basic Object Request Broker (ORB) services, as described by the CORBA specification. One instance of the ORB class exists in each client or server process at all times. An application typically accesses the ORB object using the CORBA::ORB_init function. The ORB provides methods for converting between object references (e.g., proxies) and strings, methods used to support the Dynamic Invocation Interface (DII), and initialization methods that list and retrieve references to the Naming Service, the Interface Repository, and the Basic Object Adapter (BOA).

Exceptions

```
class InvalidName : public UserException
{
public:
    static const char* exception_id;
    InvalidName () : UserException (ex_InvalidName) {}
    static InvalidName* _narrow (Exception *exception);
};
```

Types

```
typedef char* OAid;
typedef char* ObjectId;
typedef _IDL_SEQUENCE_String ObjectIdList;
```

Constants

```
static const char* ex_InvalidName;
```

ORB::_duplicate

| | |
|----------------|--|
| Overview | Duplicates an ORB object. |
| Original class | "CORBA::ORB" on page 167 |

Intended Usage

This method is intended to be used by client and server applications to duplicate a reference to an ORB object. Both the original and the duplicate reference should subsequently be released using CORBA::release(ORB_ptr).

IDL Syntax

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

Input parameters

p

The ORB object to be duplicated. The reference can be nil, in which case the return value will also be nil.

Return values

CORBA::ORB_ptr

The new ORB object reference. This value should subsequently be released using CORBA::release(ORB_ptr).

Example

```
/* For illustrative purposes, the following program
   duplicates the orb pointer */
#include "corba.h"
int main(int argc, char* argv[])
{
    int rc = 0;
    CORBA::ORB_ptr cop = CORBA::ORB_init(argc, argv, "DSOM");
    CORBA::ORB_ptr dup_cop = CORBA::ORB::_duplicate(cop);
    return rc;
}
```

ORB::_nil

| | |
|-----------------------|--|
| Overview | Returns a nil CORBA::ORB reference |
| Original class | "CORBA::ORB" on page 167 |

Intended Usage

This method is intended to be used by client and server applications to create a nil ORB reference.

IDL Syntax

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

Input parameters

None.

Return values

CORBA::ORB_ptr

A nil ORB reference.

Example

See the example in the ["Object::_nil" on page 157](#) method.

ORB::BOA_init

| | |
|-----------------------|--|
| Overview | Initializes and returns a pointer to the Basic Object Adapter (BOA) in a server. |
| Original class | "CORBA::ORB" on page 167 |
| Exceptions | "CORBA::SystemException" on page |

Intended Usage

This method is intended to be used by all server applications to both initialize the BOA and obtain a pointer to it. This method can be called multiple times without adverse effect (the BOA is only initialized once, regardless of how many times BOA_init is called). The return value should be released using CORBA::release(BOA_ptr).

IDL Syntax

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

Input parameters

argc

The number of strings in the argv array of strings. This is typically the *argc* parameter passed in to the server's main() function.

argv

An array of strings, whose size is indicated by the *argc* parameter. This is typically the *argv* parameter passed in to the server's main() function.

Note: For workstation Implementation, if one of the strings in *argv* matches *-OAid "DSOM_BOA"*, then BOA initialization is performed, the matching string is consumed, and *argc* is decremented. (The remaining strings in *argv* may be reordered as part of consuming the *-OAid "DSOM_BOA"* string.) If *argv* is NULL or contains no string that matches *-OAid "DSOM_BOA"*, then the BOA is initialized only if the *boa_identifier* parameter is *"DSOM_BOA"*.

boa_identifier

A string that indicates which BOA to initialize.

Note: For workstation Implementation, if no string in the *argv* parameter matches `-OAid "DSOM_BOA"`, then the BOA is initialized only if the *boa_identifier* parameter is "DSOM_BOA".

Return values

CORBA::BOA_ptr

A pointer to the BOA object. The return result should be released using `CORBA::release(ORB_ptr)`.

Example

```

/* This is a minimal, dummy server program. It does not create any
object to export. It assumes that the server with the name
"dummyServer" is already registered in the implementation
repository.
*/
#include #include "corba.h"
void main(int argc, char* argv[])
{
    try
    {
        /* Initialize the server's ImplementationDef, ORB, and BOA */
        CORBA::ImplRepository_ptr implrep = new CORBA::ImplRepository;
        CORBA::ImplementationDef_ptr imp =
            implrep->find_impldef_by_alias ("dummyServer");
        /* Assume op_parm is initialized. For workstation initialize
to "DSOM" */
        char * op_parm;
        /* Assume bp_parm is initialized. For workstation initialize
to "DSOM_BOA" */
        char * bp_parm;
        static CORBA::ORB_ptr op = CORBA::ORB_init(argc, argv, op_parm);
        static CORBA::BOA_ptr bp = op->BOA_init(argc, argv, bp_parm);
        bp->impl_is_ready(imp);
        /* To customize, fill in : create objects to export, and so on */
        cout << "server listening ...." << endl;
        cout.flush();
        bp->execute_request_loop(CORBA::BOA::SOMD_WAIT);
    }
    catch (CORBA::SystemException &sysdex)
    {
        cout << "caught a system exception, terminating." << endl;
        cout.flush();
    }
}

```

ORB::create_alias_tc

| | |
|-----------------------|--|
| Overview | Creates a tk_alias TypeCode. |
| Original class | "CORBA::ORB" on page 167 |
| Exceptions | "CORBA::SystemException" on page |

Intended Usage

This method is intended to be used to create a TypeCode of kind tk_alias, representing an IDL typedef.

IDL Syntax

```

CORBA::TypeCode_ptr create_alias_tc (
    CORBA::RepositoryId rep_id,
    CORBA::Identifier name,
    ORBA::TypeCode_ptr original_type);

```

Input parameters

rep_id

The Interface Repository identifier for the alias. The caller retains ownership of this string.

name

The simple name of the alias. The caller retains ownership of this string.

original_type

The non-NULL TypeCode of the type being aliased. The caller retains ownership of this TypeCode.

Return values

CORBA::TypeCode_ptr

The newly-created TypeCode. The caller assumes ownership of this TypeCode, and should subsequently release it using CORBA::release(TypeCode_ptr).

Example

```
/* Code to create a tk_alias TypeCode corresponding to this IDL
   definition: "typedef long my_long;"
*/
/* assume op initialized
*/
extern CORBA::ORB_ptr op;
CORBA::RepositoryId rep_id = CORBA::string_dup("RepositoryId_999");
CORBA::Identifier name = CORBA::string_dup("my_long");
CORBA::TypeCode_ptr tc = op->create_alias_tc (rep_id, name, CORBA::_tc_long);
```

ORB::create_array_tc

| | |
|----------------|--|
| Overview | Creates a tk_array TypeCode. |
| Original class | "CORBA::ORB" on page 167 |
| Exceptions | "CORBA::SystemException" on page |

Intended Usage

This method is intended to be used to create a TypeCode of kind tk_array, representing an IDL array.

IDL Syntax

```
CORBA::TypeCode_ptr create_array_tc (
    CORBA::ULong length,
    CORBA::TypeCode_ptr element_type_code);
```

Input parameters

length

The length of the IDL array.

element_type_code

A non-NULL TypeCode representing the type of the elements of the array. The caller retains ownership of this TypeCode.

Return values

CORBA::TypeCode_ptr

The newly-created TypeCode. The caller assumes ownership of this TypeCode, and should subsequently release it using CORBA::release(TypeCode_ptr).

Example

```
/* Code to create a tk_array TypeCode corresponding to this IDL
   definition: "typedef string my_string[1997];"
*/
/* assume op initialized */
extern CORBA::ORB_ptr op;
CORBA::TypeCode_ptr tc = op->create_array_tc(1997, CORBA::_tc_string);
```

ORB::create_context_list

| | |
|----------------|--|
| Overview | Creates a CORBA::ContextList object. |
| Original class | "CORBA::ORB" on page 167 |
| Exceptions | "CORBA::SystemException" on page |

Intended Usage

The CORBA::ORB::create_context_list method is intended to be used by client applications using the Dynamic Invocation Interface (DII), to create a CORBA::ContextList object to be subsequently passed to the CORBA::Object::create_request method.

IDL Syntax

```
CORBA::Status create_context_list (CORBA::ContextList_ptr& cntxt_list);
```

Input parameters

cntxt_list

A pointer for a CORBA::ContextList object, passed by reference, to be initialized by the CORBA::ORB::create_context_list method. The caller assumes ownership of the new ContextList object, but if the caller passes the ContextList to the CORBA::Object::create_request method, ownership of the ContextList is then transferred to the Request object.

Return values

CORBA::Status

A zero return code indicates success.

Example

```
/* The following program creates a CORBA::context list
   object and generates a system exception if appropriate
*/
#include "corba.h"
#include
int main(int argc, char* argv[])
{
    int rc = 0;
    CORBA::ContextList_ptr CLptr = CORBA::ContextList::_nil();
    /* assume op initialized */
    extern CORBA::ORB_ptr op;
    try
    {
        CORBA::Status st = orb->create_context_list(CLptr);
    }
    catch (CORBA::SystemException &se)
    {
        cout << "exception: " << se.id() << endl; rc="1";
    }
    return rc;
}
```

ORB::create_enum_tc

| | |
|----------------|--|
| Overview | Creates a tk_enum TypeCode. |
| Original class | "CORBA::ORB" on page 167 |
| Exceptions | "CORBA::SystemException" on page |

Intended Usage

This method is intended to be used to create a TypeCode of kind tk_enum, representing an IDL enum.

IDL Syntax

```
CORBA::TypeCode_ptr create_enum_tc (
    CORBA::RepositoryId rep_id,
    CORBA::Identifier name,
    CORBA::EnumMemberSeq & members);
```

Input parameters

rep_id

The non-NULL Interface Repository identifier of the IDL enum. The caller retains ownership of this string.

name

The non-NULL simple name of the IDL enum. The caller retains ownership of this string.

members

A CORBA::EnumMemberSeq object (essentially a sequence of strings) listing the members of the IDL enum. The caller retains ownership of this object. The length of this sequence cannot be zero, and the contained strings must not be NULL.

Return values

CORBA::TypeCode_ptr

The newly-created TypeCode. The caller assumes ownership of this TypeCode, and should subsequently release it using CORBA::release(TypeCode_ptr).

Example

```

/* Code to create a tk_enum TypeCode corresponding to this
   IDL definition: enum color { red, green, blue };
*/
/* assume op initialized */
extern CORBA::ORB_ptr op;
CORBA::Identifier identenum = CORBA::string_dup ("color");
CORBA::EnumMemberSeq enum_seq;
enum_seq.length(3);
enum_seq[0].type = CORBA::_tc_string;
enum_seq[0].name = CORBA::string_dup("red");
enum_seq[1].type = CORBA::_tc_string;
enum_seq[1].name = CORBA::string_dup("green");
enum_seq[2].type = CORBA::_tc_string;
enum_seq[2].name = CORBA::string_dup("blue");
CORBA::RepositoryId rep_id = CORBA::string_dup ("RepositoryId_999");
CORBA::TypeCode_ptr tc= op->create_enum_tc (rep_id, identenum, enum_seq);
...

```

ORB::create_environment

| | |
|-----------------------|--|
| Overview | Creates an Environment object. |
| Original class | "CORBA::ORB" on page 167 |
| Exceptions | "CORBA::SystemException" on page |

Intended Usage

This method is intended to be used to create an Environment object.

IDL Syntax

```
CORBA::Status create_environment (CORBA::Environment_ptr& envptr);
```

Input parameters

envptr

A pointer for a CORBA::Environment object, passed by reference, to be initialized by the CORBA::ORB::create_environment method. The caller assumes ownership of the new Environment object.

Return values

CORBA::Status

A zero return value indicates success.

Example

```

#include "corba.h"
int main(int argc, char* argv[])
{
    /* assume cop initialized */
    extern CORBA::ORB_ptr cop;
    CORBA::Environment_ptr envptr = CORBA::Environment::_nil();
    CORBA::Status status = cop->create_environment(envptr);
    return status;
}

```

ORB::create_exception_list

| | |
|-----------------------|--|
| Overview | Creates a CORBA::ExceptionList object. |
| Original class | "CORBA::ORB" on page 167 |
| Exceptions | "CORBA::SystemException" on page |

Intended Usage

The CORBA::ORB::create_exception_list method is intended to be used by client applications using the Dynamic Invocation Interface (DII), to create a CORBA::ExceptionList object to be subsequently passed to the CORBA::Object::create_request method.

IDL Syntax

```
CORBA::Status create_exception_list (CORBA::ExceptionList_ptr & excp_list);
```

Input parameters

excp_list

A pointer for a CORBA::ExceptionList object, passed by reference, to be initialized by the CORBA::ORB::create_exception_list method. The caller assumes ownership of the new ExceptionList object, but if the caller passed the ExceptionList to the

CORBA::Object::create_request method, ownership of the ExceptionList is then transferred to the Request object.

Return values
CORBA::Status

A zero return code indicates success.

Example

```
#include "corba.h"
#include
int main(int argc, char* argv[])
{
    int rc = 0;
    CORBA::ExceptionList_ptr ELptr = CORBA::ExceptionList::_nil();
    /* assume orb initialized */
    extern CORBA::ORB_ptr orb;
    try
    {
        CORBA::Status st = orb->create_exception_list(ELptr);
    }
    catch(CORBA::SystemException &se)
    {
        cout << "exception: " << se.id() << endl; rc="1;"
    }
    return rc;
}
```

ORB::create_exception_tc

| | |
|-----------------------|--|
| Overview | Creates a tk_except TypeCode. |
| Original class | "CORBA::ORB" on page 167 |
| Exceptions | "CORBA::SystemException" on page |

Intended Usage

This method is intended to be used to create a TypeCode of kind tk_except, representing an IDL exception.

IDL Syntax

```
CORBA::TypeCode_ptr create_exception_tc (
    CORBA::RepositoryId rep_id,
    CORBA::Identifier name,
    CORBA::StructMemberSeq & members);
```

Input parameters

rep_id

The non-NULL Interface Repository identifier of the IDL exception. The caller retains ownership of this string.

name

The non-NULL simple name of the IDL exception. The caller retains ownership of this string.

members

A CORBA::StructMemberSeq object (a sequence of structs of type CORBA::StructMember) listing the members of the IDL exception. Each CORBA::StructMember in the sequence specifies the name and type of the corresponding exception member; only the type member is used, and the type_def member should be set to NULL. The caller retains ownership of this object.

Return values

CORBA::TypeCode_ptr

The newly-created TypeCode. The caller assumes ownership of this TypeCode, and should subsequently release it using CORBA::release(TypeCode_ptr).

Example

```
/* Code to create a tk_except TypeCode corresponding to this IDL definition:
exception my_exception { string my_string; }'
*/
/* assume op initialized */
extern CORBA::ORB_ptr op;
```

```

CORBA::RepositoryId rep_id = CORBA::string_dup("RepositoryId_999");
CORBA::Identifier name = CORBA::string_dup("my_exception");
CORBA::StructMemberSeq st_seq;
st_seq.length(1);
st_seq[0].type = CORBA::_tc_string;
st_seq[0].name = CORBA::string_dup("my_string");
CORBA::TypeCode_ptr tc = op->create_exception_tc (rep_id, name, st_seq);

```

ORB::create_interface_tc

| | |
|-----------------------|--|
| Overview | Creates a tk_objref TypeCode. |
| Original class | "CORBA::ORB" on page 167 |
| Exceptions | "CORBA::SystemException" on page |

Intended Usage

This method is intended to be used to create a TypeCode of kind tk_objref, representing an IDL interface.

IDL Syntax

```

CORBA::TypeCode_ptr create_interface_tc (
    CORBA::RepositoryId rep_id,
    CORBA::Identifier name);

```

Input parameters

rep_id

The non-NULL Interface Repository identifier of the IDL interface. The caller retains ownership of this string.

name

The non-NULL simple name of the IDL interface. The caller retains ownership of this string.

Return values

CORBA::TypeCode_ptr

The newly-created TypeCode. The caller assumes ownership of this TypeCode, and should subsequently release it using CORBA::release(TypeCode_ptr).

Example

```

/* Code to create a tk_objref TypeCode corresponding to this
   IDL definition: interface my_interface;
   */
/* assume op initialized */
extern CORBA::ORB_ptr op;
CORBA::RepositoryId rep_id = CORBA::string_dup("RepositoryId_999");
CORBA::Identifier name = CORBA::string_dup("my_interface");
CORBA::TypeCode_ptr tc = op->create_interface_tc (rep_id, name);

```

ORB::create_list

| | |
|-----------------------|--|
| Overview | Creates a CORBA::NVList object. |
| Original class | "CORBA::ORB" on page 167 |
| Exceptions | "CORBA::SystemException" on page |

Intended Usage

This method is intended to be used to create a CORBA::NVList object, when using the Dynamic Invocation Interface (DII), to be passed to the CORBA::Object::create_request method. The caller specifies the length of the NVList to be created; upon return, the new NVList contains the specified number of (uninitialized) items. The caller must initialize the items in the NVList (or add new items) prior to passing it to the CORBA::Object::create_request method. Note, however, that since there is no mechanism for updating the flags of a NamedValue already contained by an NVList, it is advisable to create the list initially empty (that is, pass zero as the count), and then initialize the list by adding (initialized) CORBA::NamedValue objects to it, using the methods on CORBA::NVList.

See also [“ORB::create_operation_list” on page 177](#) and [“Object::_request” on page 158](#) .

IDL Syntax

```
CORBA::Status create_list (CORBA::Long count,  
                           CORBA::NVList_ptr& nvlist);
```

Input parameters

count

The number of elements in the CORBA::NVList to be created. A zero value is valid.

nvlist

A pointer for a CORBA::NVList, passed by reference, to be initialized by the CORBA::ORB::create_list method. The caller assumes ownership of the NVList object, but if the same object is passed to the CORBA::Object::create_request method, the Request object assumes ownership of the NVList.

Return values

CORBA::Status

A zero return value indicates success.

Example

```
/* The following program creates a CORBA::create_list object and  
   * generates a system exception if appropriate  
   */  
#include "corba.h"  
#include <CORBA::Long NUMITEMS = 3;  
int main(int argc, char* argv[])  
{  
    int rc = 0;  
    CORBA::NVList_ptr NVLptr = CORBA::NVList::_nil();  
    /* assume orb initialized */  
    extern CORBA::ORB_ptr orb;  
    try  
    {  
        CORBA::Status st = orb->create_list(NUMITEMS, NVLptr);  
    }  
    catch (CORBA::SystemException &se)  
    {  
        cout << "exception: " << se.id() << endl; rc="1";  
    }  
    return rc;  
}
```

ORB::create_named_value

| | |
|----------------|--|
| Overview | Creates a CORBA::NamedValue object. |
| Original class | “CORBA::ORB” on page 167 |
| Exceptions | “CORBA::SystemException” on page |

Intended Usage

The CORBA::ORB::create_named_value method is intended to be used by client applications using the Dynamic Invocation Interface (DII), to create a CORBA::NamedValue object to be subsequently passed to the CORBA::Object::create_request method.

IDL Syntax

```
CORBA::Status create_named_value (CORBA::NamedValue_ptr& nv)
```

Input parameters

nv

A pointer for a CORBA::NamedValue object, passed by reference, to be initialized by the CORBA::ORB::create_named_value method. The caller assumes ownership of the new NamedValue object, but if the caller passes the NamedValue to the CORBA::Object::create_request method, ownership of the NamedValue is then transferred to the Request object.

Return values

CORBA::Status

A zero return code indicates success.

Example

```
/* The following program creates a CORBA::NamedValue object and
   generates a system exception if appropriate
*/
#include "corba.h"
#include <int> main(int argc, char* argv[])
{
    int rc = 0;
    CORBA::NamedValue_ptr NVptr = CORBA::NamedValue::_nil();
    /* assume orb initialized */
    extern CORBA::ORB_ptr orb;
    try
    {
        CORBA::Status st = orb->create_named_value(NVptr);
    }
    catch (CORBA::SystemException &se)
    {
        cout << "exception: " << se.id() << endl; rc="1;"
    }
    return rc;
}
```

ORB::create_operation_list

| | |
|-----------------------|---|
| Overview | Creates a CORBA::NVList for a particular IDL operation. |
| Original class | "CORBA::ORB" on page 167 |
| Exceptions | "CORBA::SystemException" on page |

Intended Usage

The CORBA::ORB::create_operation_list method is intended to be used by client applications that are using the Dynamic Invocation Interface (DII), to create a CORBA::NVList object to be passed to the CORBA::Object::create_request method. The new NVList contains an item describing the name, type, and mode of each parameter of the IDL operation described by the input CORBA::OperationDef object. The application must update the NVList with the values of any in and inout parameters before invoking the corresponding DII request.

See also ["ORB::create_list" on page 175](#) .

IDL Syntax

```
CORBA::Status create_operation_list (
    CORBA::OperationDef_ptr operdf,
    CORBA::NVList_ptr& nvlist);
```

Input parameters

operdf

A non-NULL CORBA::OperationDef object, obtained from the Interface Repository, that describes the operation that the new NVList will describe. The caller retains ownership of this object.

nvlist

A pointer for a CORBA::NVList object, passed by reference, to be initialized by the CORBA::ORB::create_operation_list method. The caller assumes ownership of the new NVList object, but if the caller subsequently passes the NVList to the CORBA::Object::create_request method, the Request object then assumes ownership of the NVList.

Return values

CORBA::Status

A zero return value indicates success.

Example

See example in [CORBA::"Object::_create_request" on page 151](#) .

ORB::create_recursive_sequence_tc

| | |
|-----------------|---|
| Overview | Creates a tk_recursive_sequence TypeCode. |
|-----------------|---|

| | |
|----------------|--|
| Original class | "CORBA::ORB" on page 167 |
| Exceptions | "CORBA::SystemException" on page |

Intended Usage

This method is intended to be used to create a `TypeCode` of kind `tk_recursive_sequence`, representing a recursive IDL sequence. (A recursive sequence is one whose element type matches a type in which the recursive sequence is nested. For example, if IDL struct A contains a sequence of A, then the sequence is a recursive sequence.) The result of this method is used to construct other `TypeCodes`.

See also the `CORBA::"ORB::create_sequence_tc" on page 176` method, for creating `TypeCodes` describing non-recursive IDL sequences.

IDL Syntax

```
CORBA::TypeCode_ptr create_recursive_sequence_tc (
    CORBA::ULong bound,
    CORBA::ULong offset);
```

Input parameters

bound

The bound of the IDL sequence. Zero designates an unbounded sequence.

offset

Indicates which enclosing `TypeCode` describes the elements of the recursive sequence. It is the level of nesting of the sequence in the type that matches the sequence's elements. For example, the sequences in the following examples all have an offset of one:

```
struct foo1 {
    long value;
    sequence <foo1> chain;
};
struct foo2 {
    long value1;
    long value2;
    sequence <foo2> chain;
};
struct foo3 {
    struct foo4 {
        sequence <foo4> chain;
    };
};
```

while the sequences in the following example has an offset of two:

```
struct foo4 {
    struct foo5 {
        sequence <foo4> chain;
    };
};
```

Return values

CORBA::TypeCode_ptr

The newly-created `TypeCode`. The caller assumes ownership of this `TypeCode`., and should subsequently release it using `CORBA::release(TypeCode_ptr)`.

Example

```
/* Code to create a tk_recursive_sequence TypeCode corresponding to
   this IDL definition:
   struct my_struct { long my_long;
                     sequence my_seq; };

   */
/* assume op initialized */
extern CORBA::ORB_ptr op;
CORBA::TypeCode_ptr tc = op->create_recursive_sequence_tc (3, 1);
```

ORB::create_sequence_tc

| | |
|-----------------------|--|
| Overview | Creates a tk_sequence TypeCode. |
| Original class | "CORBA::ORB" on page 167 |
| Exceptions | "CORBA::SystemException" on page |

Intended Usage

This method is intended to be used to create a TypeCode of kind tk_sequence, representing an IDL sequence.

See also the CORBA::["ORB::create_recursive_sequence_tc" on page 177](#) method, for creating TypeCodes describing recursive IDL sequences.

IDL Syntax

```
CORBA::TypeCode_ptr create_sequence_tc (
    CORBA::ULong bound,
    CORBA::TypeCode_ptr element_type);
```

Input parameters

bound

The bound of the IDL sequence. Zero designates an unbounded sequence.

element_type

A non-NULL CORBA::TypeCode describing the type of the sequence elements. The caller retains ownership of this TypeCode.

Return values

CORBA::TypeCode_ptr

The newly-created TypeCode. The caller assumes ownership of this TypeCode, and should subsequently release it using CORBA::release(TypeCode_ptr).

Example

```
/* Code to create a tk_sequence TypeCode corresponding to this
   IDL definition:
   sequence my_seq;
*/
/* assume op initialized */
extern CORBA::ORB_ptr op;
CORBA::TypeCode_ptr tc = op->create_sequence_tc (12, CORBA::_tc_short);
```

ORB::create_string_tc

| | |
|-----------------------|--|
| Overview | Creates a tk_string TypeCode. |
| Original class | "CORBA::ORB" on page 167 |
| Exceptions | "CORBA::SystemException" on page |

Intended Usage

This method is intended to be used to create a TypeCode of kind tk_string, representing an IDL string.

IDL Syntax

```
CORBA::TypeCode_ptr create_string_tc (CORBA::ULong bound);
```

Input parameters

bound

The bound of the IDL string. Zero designates an unbounded string.

Return values

CORBA::TypeCode_ptr

The newly-created TypeCode. The caller assumes ownership of this TypeCode, and should subsequently release it using CORBA::release(TypeCode_ptr).

Example

```
/* Code to create a tk_string TypeCode corresponding to this
```

```

IDL definition:
    string <123> my_string; (this is a bounded string)
*/
/* assume op initialized */
extern CORBA::ORB_ptr op;
CORBA::TypeCode_ptr tc = op->create_string_tc (123);

```

ORB::create_struct_tc

| | |
|-----------------------|--|
| Overview | Creates a tk_struct TypeCode. |
| Original class | "CORBA::ORB" on page 167 |
| Exceptions | "CORBA::SystemException" on page |

Intended Usage

This method is intended to be used to create a TypeCode of kind tk_struct, representing an IDL struct.

IDL Syntax

```

CORBA::TypeCode_ptr create_struct_tc (
    CORBA::RepositoryId rep_id,
    CORBA::Identifier name,
    CORBA::StructMemberSeq & members);

```

Input parameters

rep_id

The non-NULL Interface Repository identifier of the IDL struct. The caller retains ownership of this string.

name

The non-NULL simple name of the IDL struct. The caller retains ownership of this string.

members

A CORBA::StructMemberSeq object (a sequence of structs of type CORBA::StructMember) listing the members of the IDL struct. Each CORBA::StructMember in the sequence specifies the name and type of the corresponding struct member; only the type member is used, and the type_def member should be set to NULL. The sequence must contain at least one CORBA::StructMember, and each CORBA::StructMember in the sequence must have a non-NULL TypeCode. The caller retains ownership of this object.

Return values

CORBA::TypeCode_ptr

The newly-created TypeCode. The caller assumes ownership of this TypeCode, and should subsequently release it using CORBA::release(TypeCode_ptr).

Example

```

/* Code to create a tk_struct TypeCode corresponding to this
IDL definition:
    struct my_struct
    {
        long my_long;
        char my_char;
    };
*/
/* assume op initialized */
extern CORBA::ORB_ptr op;
CORBA::_IDL_SEQUENCE_StructMember stm_seq;
stm_seq.length(2);
stm_seq [0].type = CORBA::_tc_long;
stm_seq [0].name = CORBA::string_dup ("my_long");
stm_seq [1].type = CORBA::_tc_char;
stm_seq [1].name = CORBA::string_dup ("my_char");
CORBA::RepositoryId rep_id = CORBA::string_dup ("RepositoryId_999");
CORBA::Identifier name = CORBA::string_dup ("my_struct");
CORBA::TypeCode_ptr tc = op->create_struct_tc (rep_id, name, stm_seq);

```

ORB::create_union_tc

| | |
|-----------------|------------------------------|
| Overview | Creates a tk_union TypeCode. |
|-----------------|------------------------------|

| | |
|----------------|--|
| Original class | "CORBA::ORB" on page 167 |
| Exceptions | "CORBA::SystemException" on page |

Intended Usage

This method is intended to be used to create a TypeCode of kind tk_union, representing an IDL union.

IDL Syntax

```
CORBA::TypeCode_ptr create_union_tc (
    CORBA::RepositoryId rep_id,
    CORBA::Identifier name,
    CORBA::TypeCode_ptr discriminator_type,
    CORBA::UnionMemberSeq & members);
```

Input parameters

rep_id

The non-NULL Interface Repository identifier of the IDL union. The caller retains ownership of this string.

name

The non-NULL simple name of the IDL union. The caller retains ownership of this string.

discriminator_type

A non-NULL CORBA::TypeCode describing the type of the union's discriminator. The caller retains ownership of this TypeCode.

members

A CORBA::UnionMemberSeq object (a sequence of unions of type CORBA::UnionMember) listing the members of the IDL union. Each CORBA::UnionMember in the sequence specifies the name, type, and member label of the corresponding union member. The type member is used, but the type_def member should be set to NULL. A union-member label of the zero octet is used to indicate the default union member. The sequence must contain at least one CORBA::UnionMember, and the TypeCode of each CORBA::UnionMember in the sequence must be non-NULL. The caller retains ownership of this object.

Return values

CORBA::TypeCode_ptr

The newly-created TypeCode. The caller assumes ownership of this TypeCode, and should subsequently release it using CORBA::release(TypeCode_ptr).

Example

```
/* Code to create a tk_union TypeCode corresponding to this
   IDL definition:
   union my_union switch (long)
   {
       case1: ulong my_ulong;
       case2: float my_float;
   };
*/
/* assume op initialized */
extern CORBA::ORB_ptr op;
CORBA::_IDL_SEQUENCE_UnionMember unmm_seq;
unmm_seq.length(2);
/* Set the member typecode and the member name for the first UnionMember */
unmm_seq [0].type = CORBA::_tc_ulong;
unmm_seq [0].name = CORBA::string_dup("my_ulong");
/* Set the member typecode and the member name for the second UnionMember */
unmm_seq [1].type = CORBA::_tc_float;
unmm_seq [1].name = CORBA::string_dup("my_float");
/* Create the Any that define the two member labels */
unmm_seq [0].label <<= (corba::long) 1;
unmm_seq [1].label <<= (CORBA::Long) 2;
corba::repositoryid rep_id="CORBA::string_dup("repositoryid_999");
" corba::identifier name="CORBA::string_dup("my_union"); "
corba::typecode_ptr discriminator_type = "CORBA::_t_long;"
corba::typecode_ptr tc="op->create_union_tc (rep_id, name,
    discriminator_type, unmm_seq);
```

ORB::get_default_context

| | |
|-----------------------|--|
| Overview | Returns the default CORBA::Context object. |
| Original class | "CORBA::ORB" on page 167 |
| Exceptions | "CORBA::SystemException" on page |

Intended Usage

This method is intended to be used by client applications to obtain a default CORBA::Context object, which can be passed to IDL operations that require a Context parameter. The default CORBA::Context object contains a name/value pair for each environment variable set in the calling process's environment.

IDL Syntax

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

Input parameters

ctx

A pointer for a CORBA::Context object, passed by reference, to be initialized by the CORBA::ORB::get_default_context method. The caller assumes ownership of the CORBA::Context object.

Return values

CORBA::Status

A zero return value indicates success.

Example

```
/* The following program creates a CORBA::Context object and generates
   system exception if appropriate
*/
#include "corba.h"
#include <int> main(int argc, char* argv[])
{
    int rc = 0;
    CORBA::Context_ptr Ctxtptr = CORBA::Context::_nil();
    /* assume orb initialized */
    extern CORBA::ORB_ptr orb;
    try
    {
        CORBA::Status st = orb->get_default_context( Ctxtptr);
    }
    catch (CORBA::SystemException &se)
    {
        cout << "exception: " << se.id() << endl; rc="1";
    }
    return rc;
}
```

ORB::get_next_response

| | |
|-----------------------|--|
| Overview | Returns the next available response, after issuing multiple deferred requests in parallel. |
| Original class | "CORBA::ORB" on page 167 |
| Exceptions | "CORBA::SystemException" on page |

Intended Usage

The CORBA::ORB::get_next_response method is intended to be used by client applications that are using the Dynamic Invocation Interface (DII), to obtain the next available response after sending multiple deferred requests in parallel (for example, using CORBA::ORB::send_multiple_requests_deferred or CORBA::Request::send_deferred). The order in which responses are received does not necessarily match the order in which requests were sent. If no response is currently available, this method will block until a response is available. To avoid blocking, use the CORBA::ORB::poll_next_reponse method.

IDL Syntax

```
CORBA::Status get_next_response (CORBA::Request_ptr& req);
```

Input parameters

req

A pointer for a CORBA::Request object, passed by reference, to be initialized by the CORBA::ORB::get_next_response method to point to the CORBA::Request object whose response was received. The CORBA::Request object is owned by the client that originally issued the Request.

Return values CORBA::Status

A zero return value indicates success.

Example

```
/* Assume the following IDL interface:
interface testObject
{
    string testMethod (in long input_value, out float out_value);
};
*/
#include "corba.h"

/* assume cop initialized */
extern CORBA::ORB_ptr cop;
/* Create the Request object */
CORBA::Object_var my_proxy = /* get a proxy somehow */
CORBA::Request_ptr req = my_proxy->request ("testMethod");
req->add_in_arg() <<= (corba::long) 12345; /* sets type and value */
corba::float out_float; req->add_out_arg() <<= out_float; /* sets type */
req->set_return_type (CORBA::tc_string);

while (!cop->poll_next_response())
{
    /* Wait */ ...
}; /* determine if a response to a deferred request is available */
cop->get_next_response(req); /* return the next available response */
...
```

ORB::get_service_information

| | |
|----------------|---|
| Overview | Describes what services of a particular type are available. |
| Original class | "CORBA::ORB" on page 167 |
| Exceptions | "CORBA::SystemException" on page |

Intended Usage

This method is intended to be used by client and server applications to determine what services of a particular type (such as Security services) are available. The result of CORBA::ORB::get_service_information does not vary during the lifetime of a single process.

IDL Syntax

```
CORBA::Boolean get_service_information (CORBA::ServiceType service_type,
CORBA::ServiceInformation& service_information);
```

Input parameters service_type

The identifier of the service for which information is needed. For example, use CORBA::Security to obtain information about what security services are available in the calling process.

service_information

A CORBA::ServiceInformation variable, passed by reference, to be initialized by the CORBA::ORB::get_service_information method.

Return values CORBA::Boolean

Zero indicates that the requested service is not available, and hence that the service_information parameter has not been updated. A nonzero return value indicates that the service_information parameter has been initialized.

Example

```
#include "corba.h"
int main(int argc, char* argv[])
{
    int rc = 0;
```

```

/* assume cop initialized */
extern CORBA::ORB_ptr cop;
CORBA::ServiceInformation si ;
/* request service information for CORBA::Security */
CORBA::Boolean retval =
    cop->get_service_information(CORBA::Security, si);
return rc;
}

```

ORB::list_initial_services

| | |
|-----------------------|---|
| Overview | Lists the runtime objects available by calling the CORBA::ORB::resolve_initial_references method. |
| Original class | "CORBA::ORB" on page 167 |
| Exceptions | "CORBA::SystemException" on page |

Intended Usage

The CORBA::ORB::list_initial_services method is intended to be used by client and server applications to determine what object references are available from the CORBA::ORB::resolve_initial_references method.

IDL Syntax

```
CORBA::ObjectIdList* list_initial_services ();
```

Input parameters

None.

Return values

CORBA::ObjectIdList *

A pointer to a sequence of strings, where each string is an identifier that can be passed to CORBA::ORB::resolve_initial_references. The caller assumes ownership of the returned result and should subsequently delete it.

Example

```

/* This program lists the runtime objects available into a
CORBA::ORB::ObjectIdList obj
*/
#include "corba.h"
#include <int> main(int argc, char* argv[])
{
    int rc = 0;
    CORBA::ORB::ObjectIdList *idlist = NULL;
    /* assume orb initialized */
    extern CORBA::ORB_ptr orb;
    try
    {
        idlist = orb->list_initial_services();
    }
    catch (CORBA::SystemException &se)
    {
        cout << "exception : " << se.id() << endl; rc="1;"
    }
    if (idlist)
    {
        /* use idlist such as idlist->length(), (*idlist)[i] where i is
        index ... */
    }
    return rc;
}

```

ORB::object_to_string

| | |
|-----------------------|---|
| Overview | Converts an object reference to an external form that can be stored outside the ORB or exchanged between processes. |
| Original class | "CORBA::ORB" on page 167 |
| Exceptions | "CORBA::SystemException" on page |

Intended Usage

The CORBA::ORB::object_to_string method is intended to be used by client or server

applications to convert object references (either proxy objects or local objects) to a string form that has meaning outside the process. The `CORBA::ORB::string_to_object` can then be used to reconstitute the object reference (either in the same process or a different process). The output string is compliant with the CORBA 2.1 specification for Interoperable Object References.

If the caller is a server (that is, the caller has already invoked `CORBA::BOA::impl_is_ready`), and the input object reference is a local object, then the resulting string can be passed to `CORBA::ORB::string_to_object` to construct a proxy in another process, or to obtain the original object pointer in the same process.

If the caller is not a server and the input object reference is a local object (rather than a proxy), then the result of `CORBA::ORB::object_to_string` is valid only within the calling process for the lifetime of the process and as long as the input object resides in the process.

IDL Syntax

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

Input parameters

obj

The object reference to be converted to string form. Nil object references are valid.

Return values

char *

The string form of the input object reference (either a proxy object or a local object). The caller assumes ownership of this string and should subsequently free it using `CORBA::string_free`.

Example

```
/* convert local object to string representation. Assume that p is
   a local object pointer already declared and defined of a class,
   say Foo
*/
#include "corba.h"
#include ...
/* assume op initialized */
extern CORBA::ORB_ptr op;
CORBA::string str = op->object_to_string(p);
CORBA::Object_ptr objPtr = op->string_to_object(str);
/* narrow down objPtr by calling Foo::_narrow(objPtr),
   get back a local obj same as p ...
*/
CORBA::string_free(str);
CORBA::release(objPtr);
...
```

ORB::poll_next_response

| | |
|-----------------------|---|
| Overview | Determines whether a response to a deferred request is available. |
| Original class | "CORBA::ORB" on page 167 |
| Exceptions | "CORBA::SystemException" on page |

Intended Usage

The `CORBA::ORB::poll_next_response` method is intended to be used by client applications that are using the Dynamic Invocation Interface (DII), to determine whether a response is available, after sending one or more deferred requests (for example, using `CORBA::ORB::send_multiple_requests_deferred` or `CORBA::Request::send`). This method can be called prior to calling `ORB::get_next_response`, to avoid blocking.

IDL Syntax

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

Input parameters

None.

Return values

CORBA::Boolean

A non-zero return value indicates that a response is available.

Example

See example in [“ORB::get_next_response” on page 182](#).

ORB::resolve_initial_references

| | |
|--|---|
| Overview | Obtains an object reference to a key service, such as the Naming Service or the Interface Repository. |
| Original class | “CORBA::ORB” on page 167 |
| Exceptions | If the input identifier is not valid, a CORBA::ORB::InvalidName exception is thrown. |
| If another error occurs, a “CORBA::SystemException” on page is thrown. | |

Intended Usage

The CORBA::ORB::resolve_initial_references method is intended to be used by client and server applications to obtain initial object references for accessing key services, such as the Interface Repository or the Naming Service. The caller specifies the identifier of the service for which a reference is needed, then narrows the return result to the proper type. For example, when the input is "InterfaceRepository", the return result should be narrowed to CORBA::Repository. When the input is "NameService", the return result is the root name context of the local naming tree., and should be narrowed to CosNaming::NamingContext (or some class derived from it). Typically an application uses CORBA::ORB::resolve_initial_references to obtain a reference to the root name context, then invokes operations on that reference to obtain all other object references.

IDL Syntax

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

Input parameters

identifier

The non-NULL identifier of the object reference to be obtained. Valid identifiers are those returned by CORBA::ORB::list_initial_services. The caller retains ownership of this string.

Return values

CORBA::Object_ptr

A reference to the requested services. The caller assumes ownership of the returned object reference, and should subsequently release it using CORBA::release.

Example

```
#include "corba.h"
...
/* assume op initialized */
extern CORBA::ORB_ptr op;
CORBA::ORB::ObjectIdList *oil = op->list_initial_services();
/* pass in the first element of oil as identifier to obtain object
reference */
CORBA::Object_ptr optr ;
optr = op->resolve_initial_references((*oil)[0]);
/* narrow optr appropriately ... */
CORBA::release(optr);
...
```

ORB::resolve_initial_references_remote

| | |
|-----------------------|--|
| Overview | Obtains an object reference to the Naming Service. |
| Original class | “CORBA::ORB” on page 167 |

| | |
|--|--|
| Exceptions | If the input identifier is not valid, a CORBA::ORB::InvalidName exception is thrown. |
| If another error occurs, a "CORBA::SystemException" on page is thrown. | |

Intended Usage

The CORBA::ORB::resolve_initial_references_remote method is intended to be used by client and server applications to obtain a reference to a NameService object from an input list of host names and associated port numbers.

The return result is the first root name context of a naming tree located from the input list of hosts. The returned object should be narrowed to CosNaming::NamingContext (or some class derived from it).

IDL Syntax

```
CORBA::Object_ptr resolve_initial_references_remote
(const char * identifier,
 const CORBA::ORB::remote_modifier host_port_list );
```

Input parameters
identifier

The non-NULL identifier of the Naming Service object reference to be obtained. This string must be "NameService".

host_port_list

This is a list of host names and associated port numbers on which the resolve_initial_references_remote operation will attempt to locate a NameService object. The operation will return the first NameService object located from the host and port combinations provided in the list.

Each string representing a hostname and port combination must be of the following syntax:

```
iiop://HostName:PortNumber
```

Return values

CORBA::Object_ptr

A reference to the requested Naming Service is returned. The caller assumes ownership of the returned object reference, and should subsequently release it using CORBA::release.

Example

```
#include "corba.h"
//-----
// - assume the ORB object pointer
// has already been initialized . . .
//-----
extern CORBA::ORB_ptr op;
CORBA::Object_ptr optr = NULL;
CORBA::String_var naming_objectid = CORBA::string_dup ("NameService");
//-----
// - create a host port list and
// provide room for three entries
//-----
CORBA::ORB::remote_modifier host_port_list;
host_port_list.length (3);
//-----
// - initialize the host port list
// with three host name and port number
// combinations . . .
//-----
host_port_list [0] = CORBA::string_dup ("iiop://hostName1:900");
host_port_list [1] = CORBA::string_dup ("iiop://hostName2:3003");
host_port_list [2] = CORBA::string_dup ("iiop://hostName3:900");
optr = op-> resolve_initial_references_remote (naming_objectid,
 host_port_list);
if (optr != NULL)
{
//-----
// - narrow the object to the appropriate object type
// - release the object (_narrow performs a _duplicate)
//-----
}
```

```

optr = CORBA::CosNaming::NamingContext::_narrow (optr);
CORBA::release (optr);
}
...

```

ORB::send_multiple_requests_deferred

| | |
|-----------------------|--|
| Overview | Issues multiple deferred requests in parallel. |
| Original class | "CORBA::ORB" on page 167 |
| Exceptions | "CORBA::SystemException" on page |

Intended Usage

The CORBA::ORB::send_multiple_requests_deferred method is intended to be used by client applications that are using the Dynamic Invocation Interface (DII), to issue multiple deferred requests in parallel. The results of these requests can later be obtained using CORBA::ORB::get_next_response. For each CORBA::Request in the input sequence, CORBA::Request::send_deferred is invoked.

IDL Syntax

```

CORBA::Status send_multiple_requests_deferred (
    const CORBA::RequestSeq& req_seq);

```

Input parameters

req_seq

A sequence of CORBA::Request_ptr objects to be invoked. An empty sequence is valid. However, each CORBA::Request_ptr object in the sequence must be non-NULL. The caller retains ownership of this sequence and of the CORBA::Request objects it contains.

Return values

CORBA::Status

A zero return value indicates that all Requests were issued.

Example

```

#include "corba.h"
...
/* assume op initialized */
extern CORBA::ORB_ptr op;
CORBA::ORB::RequestSeq reqSeq = CORBA::ORB::RequestSeq(1024);
... /* prepare each request in reqSeq */
/* issue multiple deferred requests */
CORBA::Status rc = op->send_multiple_requests_deferred(reqSeq);
...

```

ORB::send_multiple_requests_oneway

| | |
|-----------------------|--|
| Overview | Issues multiple oneway requests in parallel. |
| Original class | "CORBA::ORB" on page 167 |
| Exceptions | "CORBA::SystemException" on page |

Intended Usage

The CORBA::ORB::send_multiple_requests_oneway method is intended to be used by client applications that are using the Dynamic Invocation Interface (DII), to issue multiple oneway requests in parallel. For each CORBA::Request in the input sequence, CORBA::Request::send_oneway is invoked.

IDL Syntax

```

CORBA::Status send_multiple_requests_oneway (
    const CORBA::RequestSeq& req_seq);

```

Input parameters

req_seq

A sequence of CORBA::Request_ptr objects to be invoked. An empty sequence is valid. However, each CORBA::Request_ptr object in the sequence must be non-NULL.

The caller retains ownership of this sequence and of the CORBA::Request objects it contains.

Return values CORBA::Status

A zero return value indicates that all Requests were issued.

Example

```
#include "corba.h"
...
/* assume op initialized */
extern CORBA::ORB_ptr op;
CORBA::ORB::RequestSeq reqSeq = CORBA::ORB::RequestSeq(1024);
... /* prepare each request in reqSeq */
/* issue multiple oneway requests */
CORBA::Status rc = op->send_multiple_requests_oneway(reqSeq);
...
```

ORB::string_to_object

| | |
|-----------------------|---|
| Overview | Converts a string (produced by CORBA::ORB::object_to_string) into an object reference. |
| Original class | "CORBA::ORB" on page 167 |
| Exceptions | If the input string is not valid, or refers to a local object that is no longer valid (insofar as the ORB can determine), a "CORBA::SystemException" on page is thrown. |

Intended Usage

The CORBA::ORB::string_to_object method is intended to be used by client or server applications to convert a string form of an object reference (originally generated using CORBA::ORB::object_to_string) back into an object reference. If the input string refers to a local object residing in a server process (a process that has called CORBA::BOA::impl_is_ready), then the result is the same local object originally passed to CORBA::ORB::object_to_string. If the input string refers to a local object residing in a non-server process, then the result is the same local object originally passed to CORBA::ORB::object_to_string provided that both calls were made from the same process instance. If the input string refers to an object in another process, then CORBA::ORB::string_to_object always constructs a new proxy object. The validity of the object/server to which the proxy refers is not checked until the application invokes an application operation on the proxy.

IDL Syntax

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

Input parameters str

A string form of an object reference. This string must have been originally generated using CORBA::ORB::object_to_string (although not necessarily by the process). The caller retains ownership of this string.

Return values CORBA::Object_ptr

The object reference encoded by the input string. The caller assumes ownership of this object reference and should subsequently release it using CORBA::release.

Example

See example in [" ORB::object_to_string" on page 184](#) .

CORBA module: Policy Interface

This interface is not part of the programming model and should not be directly invoked or overridden.

CORBA module: PrimitiveDef Interface

| | |
|---|--|
| Overview | The PrimitiveDef interface is used by the Interface Repository to represent one of the OMG IDL primitive data types. |
| File name | somir.idl |
| Local-only | True |
| Ancestor interfaces | "IDLType Interface" on page 127 |
| Exceptions | "CORBA::SystemException" on page |
| Supported operations | "PrimitiveDef::kind" on page 190 |
| "IDLType::type" on page 127 | |

Intended Usage

An instance of a PrimitiveDef object is used by the Interface Repository to represent an OMG IDL primitive data type¹.

PrimitiveDef objects are not named Interface Repository objects, and as such do not reside as named objects in the Interface Repository database. PrimitiveDef objects are used to create other Interface Repository objects (both named and un-named). An instance of an PrimitiveDef object can be created using the get_primitive operation of the Repository interface.

IDL syntax

```

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, //supported on AIX and Windows NT
        pk_wchar, pk_wstring,
        pk_longdouble //not supported
    };
    interface PrimitiveDef:IDLType
    {
        readonly attribute PrimitiveKind kind;
    };
};

```

PrimitiveDef::kind

| | |
|---------------------------|---|
| Overview | The kind read operation retrieves the kind of a primitive definition (CORBA::PrimitiveDef). |
| Original interface | " PrimitiveDef Interface" on page 190 |
| Exceptions | "CORBA::SystemException" on page |

Intended Usage

The kind attribute indicates which primitive type is represented by a PrimitiveDef object. The valid values for the kind attribute that may be retrieved using the kind operation include CORBA::pk_short, CORBA::pk_long, CORBA::pk_ushort, CORBA::pk_ulong, CORBA::pk_float, CORBA::pk_double, CORBA::pk_boolean, CORBA::pk_char, CORBA::pk_wchar, and CORBA::pk_octet, that represent the basic kinds implied by the

³ The OMG IDL primitive data types include CORBA::Null, CORBA::Void, CORBA::Short, CORBA::Long, CORBA::UShort, CORBA::ULong, CORBA::Float, CORBA::Double, CORBA::Boolean, CORBA::Char, CORBA::Octet, CORBA::Any, CORBA::TypeCode, CORBA::Principal, CORBA::String, CORBA::LongLong, CORBA::ULongLong, CORBA::Wstring, CORBA::Wchar, and CORBA::Objref.

Other kind values include: CORBA::pk_any (CORBA::Any data type), CORBA::pk_TypeCode (CORBA::TypeCode data type), CORBA::pk_Principal (CORBA::Principal data type) , CORBA::pk_string (an unbounded string), CORBA::pk_wstring, and CORBA::pk_objref (CORBA::Object data type).

IDL Syntax

```
readonly attribute PrimitiveKind kind;
```

Input parameters

None.

Return values

PrimitiveKind

The returned value is the value of the kind attribute (CORBA::PrimitiveKind) of the CORBA::PrimitiveDef.

Example

```
// C++
// assume that 'this_primitive' has already been initialized
CORBA::PrimitiveDef * this_primitive;
// retrieve the 'kind' of the PrimitiveDef
CORBA::PrimitiveKind returned_kind;
returned_kind = this_primitive-> kind ();
```

CORBA module: Principal Interface

This interface is not part of the programming model and should not be directly invoked or overridden.

CORBA module: Repository Interface

| | |
|-----------------------------|--|
| Overview | The Repository interface provides global access to the Interface Repository. As it inherits from Container, it can be used to look up any definition either by the name or by id (RepositoryId). |
| File name | somir.idl |
| Local-only | True |
| Ancestor interfaces | "Container Interface" on page 85 |
| Exceptions | "CORBA::SystemException" on page |
| Supported operations | "Repository::create_array" on page 192 |
| | "Repository::create_sequence" on page 192 |
| | "Repository::create_string" on page 193 |
| | "Repository::create_wstring" on page 194 |
| | "Repository::get_primitive" on page 194 |
| | "Repository::lookup_id" on page 195 |

Intended Usage

The Repository object is a single instance object used to access member objects of the Interface Repository. The Repository object can directly contain constants (ConstantDef objects), type definitions (TypedefDef objects, including StructDef objects, UnionDef objects, EnumDef objects, and AliasDef objects), exceptions (ExceptionDef objects), interfaces (InterfaceDef objects), and modules (ModuleDef objects).

Access to the Repository object is achieved by invoking the ORB operation

resolve_initial_references.

IDL syntax

```
module CORBA {
    interface Repository:Container {
        // read interface
        Contained lookup_id (in RepositoryId search_id);
        PrimitiveDef get_primitive (in PrimitiveKind kind);
        // write interface
        StringDef create_string (in unsigned long bound);
        WstringDef create_wstring (in unsigned long bound);
        SequenceDef create_sequence(
            in unsigned long bound,
            in IDLType element_type
        );
        ArrayDef create_array (
            in unsigned long length,
            in IDLType element_type
        );
    };
};
```

Repository::create_array

| | |
|--------------------|---|
| Overview | The create_array operation is used to create a new array definition (ArrayDef). |
| Original interface | “Repository Interface” on page 191 |
| Exceptions | “CORBA::SystemException” on page |

Intended Usage

The create_array operation returns a new ArrayDef with the specified length and element_type.

IDL Syntax

```
ArrayDef create_array (
    in unsigned long length,
    in IDLType element_type
);
```

Input parameters

length

The length value specifies the length of the new ArrayDef.

element_type

The element_type is the IDLType representing each element of the ArrayDef.

Return values

ArrayDef_ptr

The return value is a pointer to the newly created ArrayDef object. The memory associated with the object is owned by the caller and can be released by invoking CORBA::release.

Example

```
// C++
// create_array
// assume that 'repository_ptr' and 'struct_1'
// have already been initialized
CORBA::Repository * repository_ptr;
CORBA::StructDef * struct_1;
// create an array definition with a bound of 409
// and array element type of 'struct_1'
CORBA::ArrayDef * array_def_ptr;
CORBA::ULong array_length = 409;
array_def_ptr = repository_ptr-> create_array (array_length, struct_1);
```

Repository::create_sequence

| | |
|--------------------|--|
| Overview | The create_sequence operation is used to create a new sequence definition (SequenceDef). |
| Original interface | “Repository Interface” on page 191 |

Intended Usage

The `create_sequence` operation returns a new `SequenceDef` with the specified bound and `element_type`.

IDL Syntax

```
SequenceDef create_sequence(
    in unsigned long bound,
    in IDLType element_type
);
```

Input parameters

bound

The bound value represents the bound of the sequence definition. The bound value can be zero.

element_type

The `element_type` is the `IDLType` of the elements in the sequence.

Return values

SequenceDef_ptr

The return value is a pointer to the `SequenceDef` of the specified bound and `element_type`.

Example

```
// C++
// assume that 'repository_ptr' and 'struct_1' have already been
// initialized
CORBA::Repository * repository_ptr;
CORBA::StructDef * struct_1;
// create a sequence of 45 'struct_1' elements . . .
CORBA::ULong bound_of_sequence = 45;
CORBA::SequenceDef * sequence_def_ptr;
sequence_def_ptr = repository_ptr-> create_sequence
    (bound_of_sequence, struct_1);
```

Repository::create_string

| | |
|--------------------|--|
| Overview | The <code>create_string</code> operation is used to create a new <code>StringDef</code> to represent a bounded string. |
| Original interface | "Repository Interface" on page 191 |
| Exceptions | "CORBA::SystemException" on page |

Intended Usage

The `create_string` operation returns a new `StringDef` with the specified bound, that must be non-zero.

Note: Unbounded strings are represented by using the ["get_primitive" on page 196](#) operation to create a `PrimitiveDef` with a kind of `CORBA::pk_string`.

IDL Syntax

```
StringDef create_string (in unsigned long bound);
```

Input parameters

bound

The bound parameter represents the bound (the maximum number of characters in the string) of the bounded string. The value must be greater than zero.

Return values

StringDef_ptr

The returned value is a pointer to a `CORBA::StringDef` object with the specified bound. The memory associated with the object is owned by the caller and can be released by invoking `CORBA::release`.

Example

```
// C++
// assume that 'repository_ptr' has already been initialized
CORBA::Repository * repository_ptr;
// create a bounded string with a bound of 51
CORBA::ULong bound_of_string = 51;
CORBA::StringDef * string_def_ptr;
string_def_ptr = repository_ptr-> create_string (bound_of_string);
```

Repository::create_wstring

| | |
|--------------------|---|
| Overview | The create_wstring operation is used to create a new WstringDef to represent a bounded wide string. |
| Original interface | "Repository Interface" on page 191 |
| Exceptions | "CORBA::SystemException" on page |

Intended Usage

The create_wstring operation returns a new WstringDef with the specified bound, that must be non-zero.

Note: Unbounded wide strings are represented by using the ["get_primitive" on page 194](#) operation to create a PrimitiveDef with a kind of CORBA::pk_wstring.

IDL Syntax

```
StringDef create_wstring (in unsigned long bound);
```

Input parameters

bound

The bound parameter represents the bound (the maximum number of wide characters in the string) of the bounded wide string. The value must be greater than zero.

Return values

StringDef_ptr

The returned value is a pointer to a CORBA::WstringDef object with the specified bound. The memory associated with the object is owned by the caller and can be released by invoking CORBA::release.

Example

```
// C++
// assume that 'repository_ptr' has already been initialized
CORBA::Repository * repository_ptr;
// create a bounded wide string with a bound of 51
CORBA::ULong bound_of_wstring = 51;
CORBA::WstringDef * wstring_def_ptr;
wstring_def_ptr = repository_ptr-> create_wstring (bound_of_wstring);
```

Repository::get_primitive

| | |
|--------------------|---|
| Overview | The get_primitive operation is used to get a PrimitiveDef object with the specified kind attribute. |
| Original interface | "Repository Interface" on page 191 |
| Exceptions | "CORBA::SystemException" on page |

Intended Usage

The get_primitive operation returns a reference to a PrimitiveDef with the specified kind attribute. All PrimitiveDefs are immutable and owned by the Repository.

IDL Syntax

```
PrimitiveDef get_primitive (in PrimitiveKind kind);
```

Input parameters

kind

The kind parameter indicates the kind of PrimitiveDef that is to be created. The valid

values for kind include CORBA::pk_null, CORBA::pk_void, CORBA::pk_short, CORBA::pk_long, CORBA::pk_ushort, CORBA::pk_ulong, CORBA::pk_float, CORBA::pk_double, CORBA::pk_longlong, CORBA::pk_ulonglong, CORBA::pk_boolean, CORBA::pk_char, CORBA::pk_wchar, CORBA::pk_octet, CORBA::pk_any, CORBA::pk_TypeCode, CORBA::pk_Principal, CORBA::pk_string, CORBA::pk_wstring, and CORBA::pk_objref.

Return values

PrimitiveDef_ptr

The return value is a pointer to the new PrimitiveDef.

Example

```
// C++
// assume 'repository_ptr' has already been initialized
CORBA::Repository * repository_ptr;
// create a PrimitiveDef to represent a CORBA::Long data type
CORBA::PrimitiveDef * pk_long_def;
pk_long_def = repository_ptr-> get_primitive (CORBA::pk_long);
```

Repository::lookup_id

| | |
|---------------------------|--|
| Overview | The lookup_id operation is used to look up an object in a Repository given its RepositoryId. |
| Original interface | "Repository Interface" on page 191 |
| Exceptions | "CORBA::SystemException" on page |

Intended Usage

The lookup_id operation is used to retrieve an object from the Interface Repository based upon its unique CORBA::RepositoryId. If the Repository does not contain a definition for the search CORBA::RepositoryId, a nil object reference is returned.

IDL Syntax

```
Contained lookup_id (in RepositoryId search_id);
```

Input parameters

search_id

The search_id parameter is the unique CORBA::RepositoryId value of the Interface Repository object that is sought.

Return values

Contained *

The returned value is a pointer to a CORBA::Contained object that was retrieved from the Interface Repository. A nil object reference is returned if no object in the Interface Repository has the specified CORBA::RepositoryId.

Example

```
// C++
// assume that 'interface_1' and 'repository_ptr' have already
// been initialized;
CORBA::InterfaceDef * interface_1;
CORBA::Repository * repository_ptr;
// obtain the CORBA::RepositoryId for 'interface_1'
CORBA::RepositoryId rep_id;
rep_id = interface_1-> id();
// . . .
// retrieve the object from the Interface Repository database
// using the CORBA::RepositoryId as the search key
CORBA::Contained * contained_ptr;
contained_ptr = repository_ptr-> lookup_id (rep_id);
```

CORBA module: Request Class

| | |
|------------------|---------------------------|
| Overview | Represents a DII request. |
| File name | request.h |

| | |
|--|---|
| Supported methods | "Request::_duplicate" on page 197 |
| "Request::_nil" on page 197 | |
| "Request::add_in_arg" on page 198 | |
| "Request::add_inout_arg" on page 198 | |
| "Request::add_out_arg" on page 199 | |
| "Request::arguments" on page 199 | |
| "Request::contexts" on page 200 | |
| "Request::ctx" on page 200 | |
| "Request::env" on page 201 | |
| "Request::exceptions" on page 201 | |
| "Request::get_response" on page 202 | |
| "Request::invoke" on page 202 | |
| "Request::operation" on page 202 | |
| "Request::poll_response" on page 203 | |
| "Request::result" on page 203 | |
| "Request::return_value" on page 204 | |
| "Request::send_deferred" on page 204 | |
| "Request::send_oneway" on page 204 | |
| "Request::set_return_type" on page 205 | |
| "Request::target" on page 205 | |

Intended Usage

The Request class provides the primary support for the Dynamic Invocation Interface (DII), which allows client applications to dynamically build and invoke requests on objects.

A Request object contains the following attributes:

target

A ["CORBA::Object" on page 150](#) object that is the target of the request.

operation

The unscoped name of the IDL operation that is executed by the request.

arguments

A ["CORBA::NVList" on page 144](#) object that describes the types of all the IDL operation's parameters, the values of the operation's in and inout parameters, and the variables in which the out parameter values are stored after the request is invoked.

result

A ["CORBA::NamedValue" on page 141](#) object that holds the result of the request after it is invoked.

env

A ["CORBA::Environment" on page 118](#) object that describes the client environment associated with the request. If an exception is raised by the request, it is reported in the client environment.

exceptions

An optional [“CORBA::ExceptionList” on page 123](#) object that describes the user-defined exceptions that the DII operation can throw. This object is essentially a list of TypeCodes for UserException subclasses.

contexts

An optional [“CORBA::ContextList” on page 103](#) object that lists the context strings that are sent with the DII operation. A ContextList object differs from a [“Context” on page 98](#) object in that a ContextList supplies only the context strings whose values are transmitted with the request, while Context is the object from which those context string values are obtained.

ctx

A [“CORBA::Context” on page 98](#) object that is passed when the request is invoked. For operations having no "context" clause in their IDL specification, this attribute is NULL.

The [“CORBA::Object::_create_request” on page 151](#) and [“CORBA::Object::_request” on page 158](#) methods can be used to create a Request object tailored to a specific IDL operation. These methods are invoked on the target object of the DII request. The `_create_request` method allows the Request object to be created and fully initialized at once. The `_request` method requires additional initialization after construction, using methods provided by the Request class. The Request class provides methods to get and set attributes, send a synchronous or asynchronous DII request, and receive the result of an asynchronous request. For additional information, see the `CORBA::Object::_create_request` and `CORBA::Object::_request` method descriptions.

Request::_duplicate

| | |
|----------------|--|
| Overview | Duplicates a Request object. |
| Original class | “CORBA::Request” on page 195 |

Intended Usage

This method is intended to be used by client and server applications to duplicate a reference to a Request object. Both the original and the duplicate reference should subsequently be released using `CORBA::release(Request_ptr)`.

IDL Syntax

```
static CORBA::Request_ptr _duplicate (CORBA::Request_ptr p);
```

Input parameters

p

The Request object to be duplicated. The reference can be nil, in which case the return value will also be nil.

Return values

CORBA::Request_ptr

The new Request object reference. This value should subsequently be released using `CORBA::release(Request_ptr)`.

Request::_nil

| | |
|----------------|--|
| Overview | Returns a nil CORBA::Request reference. |
| Original class | “CORBA::Request” on page 195 |

Intended Usage

This method is intended to be used by client and server applications to create a nil Request reference.

IDL Syntax

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

Input parameters

None.

Return values

CORBA::Request_ptr

A nil Request reference.

Request::add_in_arg

| | |
|----------------|---|
| Overview | Adds an input argument to the named value list for a DII request. |
| Original class | "CORBA::Request" on page 195 |

Intended Usage

The `add_in_arg` method is used by a client program to populate the named value list associated with a `CORBA::Request`, which was created by calling ["CORBA::Object::_request" on page 158](#). When called without a parameter, the `add_in_arg` method adds an element to the end of a ["CORBA::NVList" on page 144](#) by calling ["CORBA::NVList::add" on page 145](#) with argument passing mode `CORBA::ARG_IN`. When passed a string, the `add_in_arg` method adds an element to the end of a `CORBA::NVList` by calling ["CORBA::NVList::add_item" on page 146](#) with the input argument name and argument passing mode `CORBA::ARG_IN`.

IDL Syntax

```
CORBA::Any &add_in_arg();  
CORBA::Any &add_in_arg(const char *name);
```

Input parameters

name

The name of the argument to be added. It is legal to pass a null pointer. If specified, the input name should match the argument name specified in the IDL definition for the operation.

Return values

CORBA::Any &

The value associated with the newly created named value, to be set by the caller with the value of the input argument.

Request::add_inout_arg

| | |
|----------------|---|
| Overview | Adds an in/out argument to the named value list of a DII request. |
| Original class | "CORBA::Request" on page 195 |

Intended Usage

The `add_inout_arg` method is used by a client program to populate the named value list associated with a `CORBA::Request`, which was created by calling ["CORBA::Object::_request" on page 158](#). When called without a parameter, the `add_inout_arg` method adds an element to the end of a ["CORBA::NVList" on page 144](#) by calling ["CORBA::NVList::add" on page 145](#) with argument passing mode `CORBA::ARG_INOUT`. When passed a string, the `add_inout_arg` method adds an element to the end of a `CORBA::NVList` by calling ["CORBA::NVList::add_item" on page 146](#) with the input argument name and argument passing mode `CORBA::ARG_INOUT`.

IDL Syntax

```
CORBA::Any &add_inout_arg();
```

```
CORBA::Any &add_inout_arg(const char *name);
```

Input parameters

name

The name of the argument to be added. It is legal to pass a null pointer. If specified, the input name should match the argument name specified in the IDL definition for the operation.

Return values

CORBA::Any &

The value associated with the newly created named value, to be set by the caller with the value of the input/output argument.

Request::add_out_arg

| | |
|----------------|---|
| Overview | Adds an output argument to the named value list of a DII request. |
| Original class | "CORBA::Request" on page 195 |

Intended Usage

The `add_out_arg` method is used by a client program to populate the named value list associated with a `CORBA::Request`, which was created by calling ["CORBA::Object::_request" on page 158](#) . When called without a parameter, the `add_out_arg` method adds an element to the end of a ["CORBA::NVList" on page 144](#) by calling ["CORBA::NVList::add" on page 145](#) with argument passing mode `CORBA::ARG_OUT`. When passed a string, the `add_out_arg` method adds an element to the end of a `CORBA::NVList` by calling ["CORBA::NVList::add_item" on page 146](#) with the input argument name and argument passing mode `CORBA::ARG_OUT`.

IDL Syntax

```
CORBA::Any &add_out_arg();  
CORBA::Any &add_out_arg(const char *name);
```

Input parameters

name

The name of the argument to be added. It is legal to pass a null pointer. If specified, the input name should match the argument name specified in the IDL definition for the operation.

Return values

CORBA::Any &

The value associated with the newly created named value, to be set by the caller with the value of the output argument.

Request::arguments

| | |
|----------------|---|
| Overview | Retrieves the argument list of a DII request. |
| Original class | "CORBA::Request" on page 195 |

Intended Usage

The `arguments` method is used by a client program to access the argument list of a Dynamic Invocation Interface (DII) request. The argument list is specified using a ["CORBA::NVList" on page 144](#) object and describes the types of all the IDL operation's parameters, the values of the operation's in and inout parameters, and the variables in which the out parameter values are stored after the DII request is invoked. For additional information, see the `NVList` class description. The argument list of a `Request` object is set by the ["CORBA::Object::_create_request method" on page 151](#) .

IDL Syntax

```
CORBA::NVList_ptr arguments();
```

Input parameters

None.

Return values

CORBA::NVList_ptr

A pointer to the argument list of the DII request, if any, or a null pointer. Ownership of the return value is maintained by the Request object; the return value must not be freed by the caller. If the DII request raises an exception, the values of the operation's out parameters are unpredictable.

Request::contexts

| | |
|----------------|---|
| Overview | Accesses the list of context strings that is sent with a DII request. |
| Original class | "CORBA::Request" on page 195 |

Intended Usage

The contexts method is used by a client application to access the list of context strings that are sent with a Dynamic Invocation Interface (DII) request, as optionally input to the ["CORBA::Object::_create_request" on page 151](#) method. The context list is specified using a ["CORBA::ContextList" on page 103](#) object and is used to improve performance. When invoking a request without an associated ContextList object, the ORB looks up context information in the Interface Repository. For additional information, see the ContextList class description.

IDL Syntax

```
CORBA::ContextList_ptr contexts();
```

Input parameters

None.

Return values

CORBA::ContextList_ptr

A pointer to the list of context strings that are sent with the DII request, as input to ["CORBA::Object::_create_request" on page 151](#), or a null pointer. Ownership of the return result is maintained by the Request object; the return value must not be freed by the caller.

Request::ctx

| | |
|----------------|---|
| Overview | Gets and sets the Context object associated with a DII request. |
| Original class | "CORBA::Request" on page 195 |

Intended Usage

The ctx method is used by a client application to get and set the list of properties that are sent with a Dynamic Invocation Interface (DII) request. The list of properties is specified using a ["CORBA::Context" on page 98](#) object and is used to pass information from the client environment to the server environment. For additional information, see the Context class description. The Context object associated with a DII request can also be set by the ["CORBA::Object::_create_request" on page 151](#) method.

IDL Syntax

```
void ctx(CORBA::Context_ptr p);  
CORBA::Context_ptr ctx() const;
```

Input parameters

new_context

A pointer to the new Context object to be associated with the DII request. If a Context

object is already associated with the request, it is released. The caller retains ownership of this parameter (the Request object makes a duplicate). It is valid to pass a null pointer.

Return values

CORBA::Context_ptr

A pointer to the Context currently associated with the DII request, if any, or a null pointer. Ownership of the return value is maintained by the Request object; the return value must not be freed by the caller.

Request::env

| | |
|-----------------------|---|
| Overview | Retrieves the client environment associated with a DII request. |
| Original class | "CORBA::Request" on page 195 |

Intended Usage

The env method is used by a client application to access an exception raised by a Dynamic Invocation Interface (DII) request. The exception is held in a ["CORBA::Environment" on page 118](#) object, which is used for error handling in those cases where catch/throw exception handling cannot be used (such as DII). For additional information, see the Environment class description. The Environment object is automatically created by the ["CORBA::Object::_create_request" on page 151](#) or ["CORBA::Object::_request" on page 158](#) method used to construct the Request object.

IDL Syntax

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

Input parameters

None.

Return values

CORBA::Environment_ptr

A pointer to the client environment associated with a DII request. Ownership of the return value is maintained by the Request object; the return value must not be freed by the caller.

Request::exceptions

| | |
|-----------------------|--|
| Overview | Retrieves the list of user-defined exceptions that can be thrown by a DII request. |
| Original class | "CORBA::Request" on page 195 |

Intended Usage

The exceptions method is used by a client application to access the list of user-defined exceptions that can be thrown by a Dynamic Invocation Interface (DII) request, as optionally input to ["CORBA::Object::_create_request" on page 151](#). The exception list is specified using a ["CORBA::ExceptionList" on page 123](#) object and is used to improve performance. When invoking a request without an associated ExceptionList object, the ORB looks up user-defined exception information in the Interface Repository. For additional information, see the ExceptionList class description.

IDL Syntax

```
CORBA::ExceptionList_ptr exceptions();
```

Input parameters

None.

Return values

CORBA::ExceptionList_ptr

A pointer to the list of user-defined exceptions that can be thrown by the DII request, as input to [“CORBA::Object::_create_request” on page 151](#) , or a null pointer. Ownership of the return result is maintained by the Request object; the return value must not be freed by the caller.

Request::get_response

| | |
|----------------|---|
| Overview | Get the response from the request that is expected after a send_deferred has been issued. |
| Original class | “CORBA::Request” on page 195 |
| Exceptions | “CORBA::SystemException” on page |

Intended Usage

Use of the get_response method will result in a blocking call if the response hasn't been received yet. For client DII applications, use of the [“ORB::get_next_response” on page 182](#) method is recommended.

IDL Syntax

```
CORBA::Status get_response();
```

Input parameters

None.

Return values

CORBA::Status

A zero return code indicates the response was successfully received. A non-zero return code indicates failure.

Request::invoke

| | |
|----------------|--|
| Overview | Sends a synchronous DII request. |
| Original class | “CORBA::Request” on page 195 |

Intended Usage

The invoke method is used by a client application that is using the Dynamic Invocation Interface (DII), to issue a request. The invoke method blocks until a response is received. A Request object is constructed using the [“CORBA::Object::_create_request” on page 151](#) or [“CORBA::Object::_request” on page 158](#) method.

IDL Syntax

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

Input parameters

None.

Return values

CORBA::Status

A zero return code indicates the DII request was successfully sent and the response received. A non-zero return code indicates failure.

Request::operation

| | |
|----------------|---|
| Overview | Retrieves the unscoped operation name of a DII request. |
| Original class | “CORBA::Request” on page 195 |

Intended Usage

The operation method is used by a client application to access the unscoped operation name of a Dynamic Invocation Interface (DII) request. The operation name of a Request object is set by either the [“CORBA::Object::_create_request” on page 151](#) or [“CORBA::Object::_request” on page 158](#) method.

IDL Syntax

```
const char * operation();
```

Input parameters

None.

Return values

const char *

The unscoped operation name of the DII request. Ownership of the return value is maintained by the Request object; the return value must not be freed by the caller.

Request::poll_response

| | |
|-----------------------|--|
| Overview | Determines whether a response to an asynchronous request is available. |
| Original class | “CORBA::Request” on page 195 |

Intended Usage

The poll_response method is used by a client application that is using the Dynamic Invocation Interface (DII), to determine whether a response is available, after sending one or more deferred requests (for example, using [“CORBA::Request::send_deferred” on page 204](#)). This method can be called prior to calling [“CORBA::Request::get_response” on page 202](#) , to avoid blocking.

IDL Syntax

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

Input parameters

None.

Return values

CORBA::Boolean

A zero return value indicates that a response is not available. A non-zero return value indicates that a response has been received.

Request::result

| | |
|-----------------------|--|
| Overview | Retrieves the return value of a DII request. |
| Original class | “CORBA::Request” on page 195 |

Intended Usage

The result method is used by a client application to access the return value of a Dynamic Invocation Interface (DII) request. The return value is specified using a [“CORBA::NamedValue” on page 141](#) object and must not be accessed until after the request has been invoked. For additional information, see the NamedValue class description. The return value of a Request object is set by the [“CORBA::Request::invoke” on page 202](#) or [“CORBA::Request::get_response” on page 202](#) method, depending whether the request is synchronous or asynchronous, respectively.

IDL Syntax

```
CORBA::NamedValue_ptr result();
```

Input parameters

None.

Return values

CORBA::NamedValue_ptr

A pointer to the return value of the DII request. Ownership of the return value is maintained by the Request object; the return value must not be freed by the caller. If the DII request raises an exception, the return value is unpredictable.

Request::return_value

| | |
|----------------|--|
| Overview | Returns the value of the return type of a DII request. |
| Original class | "CORBA::Request" on page 195 |

Intended Usage

The return_value method is used by a client program to access the value of the return type of a DII request. Specifically, the return_value method returns the ["CORBA::Any" on page 55](#) contained in the ["CORBA::NamedValue" on page 141](#) holding the return value of a DII request.

IDL Syntax

```
CORBA::Any &return_value();
```

Input parameters

None.

Return values

CORBA::Any &

The value of the return type of the DII request.

Request::send_deferred

| | |
|----------------|--|
| Overview | Sends an asynchronous DII request. |
| Original class | "CORBA::Request" on page 195 |

Intended Usage

The send_deferred method is used by a client application that is using the Dynamic Invocation Interface (DII), to issue an asynchronous request. The results of an asynchronous request are later obtained using ["CORBA::Request::get_response" on page 202](#). The ["CORBA::Request::poll_response" on page 203](#) method is used to determine whether a response is available. A Request object is constructed using the ["CORBA::Object::_create_request" on page 151](#) or ["CORBA::Object::_request" on page 158](#) method.

IDL Syntax

```
CORBA::Status send_deferred();
```

Input parameters

None.

Return values

CORBA::Status

A zero return code indicates the DII request was successfully sent. A non-zero return code indicates failure.

Request::send_oneway

| | |
|-----------------------|--|
| Overview | Sends a oneway DII request. |
| Original class | "CORBA::Request" on page 195 |

Intended Usage

The `send_oneway` method is used by a client application that is using the Dynamic Invocation Interface (DII), to issue a oneway request. No response is sent back, so the client application must not call ["CORBA::Request::get_response" on page 202](#) . A Request object is constructed using the ["CORBA::Object::_create_request" on page 151](#) or ["CORBA::Object::_request" on page 158](#) method.

IDL Syntax

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

Input parameters

None.

Return values

CORBA::Status

A zero return code indicates the DII request was successfully sent. A non-zero return code indicates failure.

Request::set_return_type

| | |
|-----------------------|--|
| Overview | Sets the return type for a DII request. |
| Original class | "CORBA::Request" on page 195 |

Intended Usage

The `set_return_type` method is used by a client program to set the return type for a `CORBA::Request`, which was created by calling ["CORBA::Object::_request" on page 158](#) .

IDL Syntax

```
CORBA::Void set_return_type(CORBA::TypeCode_ptr tc);
```

Input parameters

tc

A pointer to a ["CORBA::TypeCode" on page 221](#) representing the type of the operation return value. The caller retains ownership of this parameter (the `CORBA::Request` makes its own copy).

Return values

None.

Request::target

| | |
|-----------------------|--|
| Overview | Retrieves a pointer to the target object of a DII request. |
| Original class | "CORBA::Request" on page 195 |

Intended Usage

The `target` method is used by a client program to access the target object of a DII request. The target object of a Request object is automatically set by the ["CORBA::Object::_create_request" on page 151](#) or ["CORBA::Object::_request" on page 158](#) method. These methods are invoked on the target object of the DII request.

IDL Syntax

```
CORBA::Object_ptr target() const;
```

Input parameters

None.

Return values

CORBA::Object_ptr

A pointer to the target object of the request. Ownership of the return value is maintained by the Request; the return value must not be freed by the caller.

CORBA module: RequestSeq Class

| | |
|--------------------------|---|
| Overview | Specifies a list of Requests. |
| File name | orb.h |
| Supported methods | "RequestSeq::allocbuf" on page 206 |
| | "RequestSeq::freebuf" on page 206 |
| | "RequestSeq::length" on page 207 |
| | "RequestSeq::maximum" on page 207 |
| | "RequestSeq::operator[]" on page 208 |

Intended Usage

Specifies a list of Requests to be sent in parallel using the Dynamic Invocation Interface. RequestSeq is used to specify the collection of requests sent by the ORB methods ["send_multiple_requests_oneway" on page 188](#) and ["send_multiple_requests_deferred" on page 188](#) . For additional information, see the ["Request" on page 195](#) and ["ORB" on page 167](#) class descriptions.

RequestSeq::allocbuf

| | |
|-----------------------|---|
| Overview | Allocates a sequence of pointers to Request elements. |
| Original class | "CORBA::RequestSeq" on page 206 |

Intended Usage

The allocbuf method is used by a client program to allocate a sequence of pointers to Request elements. The pointers are initialized to NULL. The newly allocated buffer can be passed to the "CORBA::Request_ptr *" constructor. Memory allocated using the allocbuf method must be freed using the freebuf method or by transferring ownership to a RequestSeq object.

IDL Syntax

```
CORBA::Request_ptr allocbuf(CORBA::ULong nelems);
```

Input parameters

nelems

The number of pointers to Request elements to be allocated. The requested number of elements must be greater than zero.

Return values

CORBA::Request_ptr *

A pointer to the address of the newly allocated sequence of pointers to Request elements. Ownership of the return value transfers to the caller. If the allocbuf method fails, a null pointer is returned.

RequestSeq::freebuf

| | |
|-----------------------|---|
| Overview | Frees a sequence of Request elements. |
| Original class | "CORBA::RequestSeq" on page 206 |

Intended Usage

The freebuf method is used by a client program to free a sequence of Request elements. The release method is called on each Request element.

IDL Syntax

```
void freebuf(CORBA::Request_ptr *data);
```

Input parameters

data

A pointer to the sequence of Request elements to be freed. The freebuf method ignores null pointers passed to it.

Return values

None.

RequestSeq::length

| | |
|-----------------------|---|
| Overview | Gets and sets the number of Request elements in a sequence. |
| Original class | "CORBA::RequestSeq" on page 206 |

Intended Usage

The length method is used by a client program to get and set the number of Requests in a collection of DII requests. Increasing the number of Request elements causes the sequence buffer to be reallocated. Decreasing the number of Request elements causes the orphaned Requests to be released.

IDL Syntax

```
CORBA::ULong length() const;
void length(CORBA::ULong len);
```

Input parameters

len

The desired number of Request elements in the sequence.

Return values

CORBA::ULong

The current number of Request elements in the sequence.

RequestSeq::maximum

| | |
|-----------------------|---|
| Overview | Retrieves the maximum number of Request elements in a sequence. |
| Original class | "CORBA::RequestSeq" on page 206 |

Intended Usage

The maximum method is used by a client program when querying the RequestSeq object associated with a collection of DII requests. For an unbounded sequence, the maximum method returns the number of Request elements currently allocated in a sequence. This tells applications the total amount of buffer space available. The application can then determine how many additional Request elements can be inserted before the buffer is reallocated. For a bounded sequence, the maximum method returns the maximum number of Request elements in a sequence as specified in the IDL type declaration. By definition, the maximum number of Request elements in a bounded sequence cannot be changed.

IDL Syntax

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

Input parameters

None.

Return values

CORBA::ULong

The maximum number of Request elements in the sequence.

RequestSeq::operator[]

| | |
|-----------------------|---|
| Overview | Retrieves the Request element at the specified index. |
| Original class | "CORBA::RequestSeq" on page 206 |
| Exceptions | "CORBA::SystemException" on page |

Intended Usage

The subscript operator is called by a client program when querying the RequestSeq object associated with a collection of DII requests. The subscript operator returns the Request element at the specified index. Const and non-const versions of this operator are provided.

IDL Syntax

```
CORBA::Request_SeqElem operator[] (CORBA::ULong index);  
CORBA::Request_SeqElem operator[] (CORBA::ULong index) const;
```

Input parameters

index

The index corresponding to the desired Request element, starting at zero. A system exception is raised if the input index is greater than or equal to the number of elements in the sequence.

Return values

CORBA::Request_SeqElem

The Request element at the specified index. Ownership of the return value does not transfer to the caller. However, the sequence buffer may be owned by the caller prior to the method invocation.

CORBA module: SequenceDef Interface

| | |
|-----------------------------|---|
| Overview | A SequenceDef object represents an OMG IDL sequence definition in the Interface Repository. |
| File name | somir.idl |
| Local-only | True |
| Ancestor interfaces | "IDLType Interface" on page 127 |
| Exceptions | "CORBA::SystemException" on page |
| Supported operations | "SequenceDef::bound" on page 209 |
| | "SequenceDef::element_type" on page 209 |
| | "SequenceDef::element_type_def" on page 210 |
| | "IDLType::type" on page 127 |

Intended Usage

An instance of a SequenceDef object is used by the Interface Repository to represent an OMG IDL bounded sequence data type.

SequenceDef objects are not named Interface Repository objects, and as such do not reside as named objects in the Interface Repository database (they are in a group of interfaces known as Anonymous types). An instance of a SequenceDef object can be created using the [“create_sequence” on page 192](#) operation of the Repository interface.

IDL syntax

```
module CORBA {
    interface SequenceDef:IDLType {
        attribute unsigned longbound;
        readonlyattribute TypeCode element_type;
        attribute IDLType element_type_def;
    };
};
```

SequenceDef::bound

| | |
|---------------------------|---|
| Overview | The bound read and write operations allow the access and update of the bound attribute of a sequence definition (CORBA::SequenceDef) within the Interface Repository. |
| Original interface | “SequenceDef Interface” on page 208 |
| Exceptions | “CORBA::SystemException” on page |

Intended Usage

The bound attribute specifies the maximum number of elements in the sequence. A bound of zero indicates an unbounded sequence. Read and write bound operations are supported with parameters as defined below.

IDL Syntax

```
attribute unsigned longbound;
```

Read operations

Input parameters

None.

Return values

CORBA::ULong

The returned value is the current value of the bound attribute of the sequence definition (CORBA::SequenceDef) object.

Write operations

Input parameters

CORBA::ULong bound

The bound parameter is the new value to which the bound attribute of the CORBA::SequenceDef object is set.

Return values

None.

Example

```
// C++
// assume that 'this_sequence' has already been initialized
CORBA::SequenceDef * this_sequence;
// change the bound attribute of the sequence definition
CORBA::ULong new_bound = 409;
this_sequence-> bound (new_bound);
// obtain the bound of a sequence definition
CORBA::ULong returned_bound;
returned_bound = this_sequence-> bound ();
```

SequenceDef::element_type

| | |
|-----------------|---|
| Overview | The element_type operation returns a type |
|-----------------|---|

| | |
|---------------------------|--|
| | (CORBA::TypeCode *) representative of the sequence element of a SequenceDef. |
| Original interface | "SequenceDef Interface" on page 208 |
| Exceptions | "CORBA::SystemException" on page |

Intended Usage

The element_type attribute of a SequenceDef object references a CORBA::TypeCode * that represents the type of the sequence element. The element_type read operation returns a copy of the CORBA::TypeCode referenced by the element_type attribute.

IDL Syntax

```
readonlyattribute TypeCode element_type;
```

Input parameters

None.

Return values

TypeCode *

The returned value is a pointer to a copy of the CORBA::TypeCode referenced by the element_type attribute. The memory is owned by the caller and can be returned by invoking CORBA::release.

Example

```
// C++
// assume that 'this_sequence' has already been initialized
CORBA::SequenceDef * this_sequence;
// retrieve the TypeCode which represents the type of the sequence
// elements
CORBA::TypeCode * sequence_element_type;
sequence_element_type = this_sequence-> element_type
```

SequenceDef::element_type_def

| | |
|---------------------------|--|
| Overview | The element_type_def read and write operation allow the access and update of the element type definition of a sequence definition (SequenceDef) in the Interface Repository. |
| Original interface | "SequenceDef Interface" on page 208 |
| Exceptions | "CORBA::SystemException" on page |

Intended Usage

The type of the elements within a sequence definition is identified by the element_type_def attribute (a reference to a CORBA::IDLType *). Setting the element_type_def attribute also updates the element_type attribute as well as the inherited type attribute.

IDL Syntax

```
attribute IDL/Type element_type_def;
```

Read operations

Input parameters

None.

Return values

CORBA::IDLType_ptr

The returned object is a pointer to a copy of the IDLType referenced by the element_type_def attribute of the SequenceDef object. The returned object is owned by the caller and can be released using CORBA::release.

Write operations

Input parameters

CORBA::IDLType_ptr element_type_def

The element_type_def parameter represents the new sequence element definition for the SequenceDef.

Return values

None.

Example

```
// C++
// assume that 'this_sequence' and 'this_union' have already been
// initialized
CORBA::SequenceDef * this_sequence;
CORBA::UnionDef * this_union;
// change the sequence element type definition to 'this_union'
this_sequence-> element_type_def (this_union);
// read the element type definition from 'this_sequence'
CORBA::IDLType * returned_element_type_def;
returned_element_type_def = this_sequence-> element_type_def ();
```

CORBA module: ServerRequest Class

| | |
|--------------------------|--|
| Overview | Provides information about a request to be dispatched by a BOA::DynamicImplementation. |
| File name | request.h |
| Supported methods | "ServerRequest::_duplicate" on page 211 |
| | "ServerRequest::_nil" on page 212 |
| | "ServerRequest::ctx" on page 212 |
| | "ServerRequest::exception" on page 213 |
| | "ServerRequest::op_def" on page 213 |
| | "ServerRequest::op_name" on page 213 |
| | "ServerRequest::params" on page 214 |
| | "ServerRequest::result" on page 215 |

Intended Usage

The ServerRequest class is intended to be used by an implementation of a subclass of ["CORBA::BOA::DynamicImplementation" on page 75](#) , within the ["invoke" on page 75](#) method. The ServerRequest class is part of the DynamicSkeleton Interface (DSI), used primarily to create inter-ORB bridges or gateway servers.

The ServerRequest object provides information to the CORBA::BOA::DynamicImplementation ::invoke method about the operation to be invoked and the in and inout parameter values. It also provides methods for recording the output and return values after the operation has been dispatched, so that the response can be sent back to the calling client.

ServerRequest::_duplicate

| | |
|-----------------------|--|
| Overview | Duplicates a ServerRequest object. |
| Original class | "CORBA::ServerRequest" on page 211 |

Intended Usage

This method is intended to be used by client and server applications to duplicate a reference to a ServerRequest object. The duplicate reference should subsequently be released using CORBA::release(ServerRequest_ptr).

IDL Syntax

```
static CORBA::ServerRequest_ptr _duplicate (CORBA::ServerRequest_ptr p);
```

Input parameters

p

The ServerRequest object to be duplicated. The reference can be nil, in which case the return value will also be nil.

Return values

CORBA::ServerRequest_ptr

The new ServerRequest object reference. This value should subsequently be released using CORBA::release(ServerRequest_ptr).

ServerRequest::_nil

| | |
|----------------|--|
| Overview | Returns a nil CORBA::ServerRequest reference. |
| Original class | "CORBA::ServerRequest" on page 211 |

Intended Usage

This method is intended to be used by client and server applications to create a nil ServerRequest reference.

IDL Syntax

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

Input parameters

None.

Return values

CORBA::ServerRequest_ptr

A nil ServerRequest reference.

ServerRequest::ctx

| | |
|----------------|---|
| Overview | Provides the Context of an operation being invoked on a BOA::DynamicImplementation. |
| Original class | "CORBA::ServerRequest" on page 211 |
| Exceptions | "CORBA::SystemException" on page |

Intended Usage

This method is intended to be used by an implementation of ["CORBA::BOA::DynamicImplementation::invoke" on page 75](#) (in a subclass of BOA::DynamicImplementation) to discover the Context of the request (if any). The IDL specification (for the operation being dispatched by CORBA::BOA::DynamicImplementation::invoke) indicates what Context identifiers are transmitted on each invocation of that operation.

IDL Syntax

```
CORBA::Context_ptr ctx() throw (CORBA::SystemException);
```

Input parameters

None.

Return values

CORBA::Context_ptr

The Context of the operation that an implementation of CORBA::BOA::DynamicImplementation::invoke is dispatching. The ServerRequest

retains ownership of the Context and the caller must not modify or free it.

ServerRequest::exception

| | |
|----------------|--|
| Overview | Stores an exception in a ServerRequest. |
| Original class | "CORBA::ServerRequest" on page 211 |
| Exceptions | "CORBA::SystemException" on page |

Intended Usage

This method is intended to be called from an implementation of ["CORBA::BOA::DynamicImplementation::invoke" on page 75](#) (in a subclass of BOA::DynamicImplementation) when an exception has been thrown by the operation being dispatched by the CORBA::BOA::DynamicImplementation::invoke method. This method can be called at most once by an execution of CORBA::BOA::DynamicImplementation::invoke, and only after ["CORBA::ServerRequest::params" on page 214](#) has been called. ServerRequest::exception may not be called if ["CORBA::ServerRequest::result" on page 215](#) has already been called. The ServerRequest object assumes ownership of the input Any object.

IDL Syntax

```
void exception (CORBA::Any *value)
    throw (CORBA::SystemException);
```

Input parameters

value

A CORBA::Any containing the exception to be stored in the ServerRequest. This exception is sent back to the client that originated the request. The ServerRequest assumes ownership of this Any.

Return values

None.

ServerRequest::op_def

| | |
|----------------|---|
| Overview | Describes the signature of an operation being invoked on a DynamicImplementation. |
| Original class | "CORBA::ServerRequest" on page 211 |
| Exceptions | "CORBA::SystemException" on page |

Intended Usage

This method is intended to be used by an implementation of ["CORBA::BOA::DynamicImplementation::invoke" on page 75](#) (in a subclass of DynamicImplementation), to discover the signature of an operation being dispatched.

IDL Syntax

```
CORBA::OperationDef_ptr op_def()
    throw (CORBA::SystemException);
```

Input parameters

None.

Return values

CORBA::OperationDef_ptr

A pointer to the OperationDef object from the Interface Repository that describes the operation being dispatched. The caller assumes ownership of this object.

ServerRequest::op_name

| | |
|-----------------------|--|
| Overview | Indicates the name of an operation being invoked on a DynamicImplementation. |
| Original class | "CORBA::ServerRequest" on page 211 |
| Exceptions | "CORBA::SystemException" on page |

Intended Usage

This method is intended to be used by an implementation of ["CORBA::BOA::DynamicImplementation::invoke" on page 75](#) (in a subclass of DynamicImplementation), to discover which operation needs to be dispatched.

IDL Syntax

```
CORBA::Identifier op_name()
    throw (CORBA::SystemException);
```

Input parameters

None.

Return values

Identifier (char *)

The (unscoped) IDL name of the operation being dispatched by an implementation of ["CORBA::BOA::DynamicImplementation::invoke" on page 75](#) . The ServerRequest retains ownership of this string and the caller must not modify it. For attribute accessor methods, the operation names are `_get_<attribute>` and `_set_<attribute>`. For operations introduced in CORBA::Object, the operation names are `_interface`, `_implementation`, `_is_a`, and `_non_existent`.

ServerRequest::params

| | |
|-----------------------|---|
| Overview | Retrieves the in and inout parameter values of a ServerRequest. |
| Original class | "CORBA::ServerRequest" on page 211 |
| Exceptions | "CORBA::SystemException" on page |

Intended Usage

This method is intended to be used by an implementation of ["CORBA::BOA::DynamicImplementation::invoke" on page 75](#) (in a subclass of DynamicImplementation), to discover the the in and inout parameter values for the operation being dispatched. The caller supplies the types of the parameters via an NVList, and receives the parameter values in the same NVList. An implementation of CORBA::BOA::DynamicImplementation::invoke must invoke CORBA::ServerRequest::params exactly once.

IDL Syntax

```
void params (CORBA::NVList_ptr parameters)
    throw (CORBA::SystemException);
```

Input parameters

parameters

An NVList containing the TypeCodes (but not the values) for the parameters of the method being dispatched.

Return values

parameters

On output, the NVList additionally contains the values of any in and inout parameters for that operation. This same NVList should subsequently be modified by the invoke() method (after dispatching the target method) to record the output parameter values.

The ServerRequest assumes ownership of the NVList. If the operation has no parameters, an empty NVList can be passed.

ServerRequest::result

| | |
|-----------------------|--|
| Overview | Records the return value of an operation invoked on a DynamicImplementation. |
| Original class | "CORBA::ServerRequest" on page 211 |
| Exceptions | "CORBA::SystemException" on page |

Intended Usage

This method is intended to be used by an implementation of ["CORBA::BOA::DynamicImplementation::invoke" on page 75](#) (in a subclass of DynamicImplementation), to record the return result of an operation that was dispatched. If there is no return value (the return type is void or an exception occurred), CORBA::ServerRequest::result should not be called. The CORBA::ServerRequest::result method can be called at most once by an execution of CORBA::BOA::DynamicImplementation::invoke, and only after calling ["CORBA::ServerRequest::params" on page 214](#) .

IDL Syntax

```
void result (CORBA::Any *value)
    throw (CORBA::SystemException);
```

Input parameters

value

A CORBA::Any containing the return result of the operation invoked by an implementation of ["CORBA::BOA::DynamicImplementation::invoke" on page 75](#) . The ServerRequest assumes ownership of the Any.

Return values

None.

CORBA module: StringDef Interface

| | |
|-----------------------------|--|
| Overview | The StringDef interface is used to represent an OMG IDL bounded string type. |
| File name | somir.idl |
| Local-only | True |
| Ancestor interfaces | "IDLType Interface" on page 127 |
| Exceptions | "CORBA::SystemException" on page |
| Supported operations | "StringDef::bound" on page 217 |

Intended Usage

An instance of a StringDef object is used by the Interface Repository to represent an OMG IDL bounded string data type.

The StringDef object is not a named Interface Repository object (it is in a group of interfaces known as Anonymous types), and as such does not reside as a named object in the Interface Repository database. An instance of a StringDef object can be created using the ["create_string" on page 193](#) operation of the Repository interface.

IDL syntax

```
module CORBA
{
```

```

interface StringDef:IDLType
{
    attribute unsigned longbound;
};
};

```

StringDef::bound

| | |
|---------------------------|---|
| Overview | The bound read and write operations allow the access and update of the bound attribute of a bounded string definition (CORBA::StringDef) within the Interface Repository. |
| Original interface | "StringDef Interface" on page 215 |
| Exceptions | "CORBA::SystemException" on page |

Intended Usage

The bound attribute specifies the maximum number of characters in the string, and must not be zero.

IDL Syntax

```
attribute unsigned longbound;
```

Read operations

Input parameters

None.

Return values

CORBA::ULong

The returned value is the current value of the bound attribute of the string definition (CORBA::StringDef) object.

Write operations

Input parameters

CORBA::ULong bound

The bound parameter is the new value to which the bound attribute of the CORBA::StringDef object is set.

Return values

None.

Example

```

// C++
// assume that 'this_string' has already been initialized
CORBA::StringDef * this_string;
// change the bound attribute of the string definition
CORBA::ULong new_bound = 409;
this_string-> bound (new_bound);
// obtain the bound of a string definition
CORBA::ULong returned_bound;
returned_bound = this_string-> bound ();

```

CORBA module: StructDef Interface

| | |
|----------------------------|--|
| Overview | The StructDef interface is used to represent and OMG IDL structure definition. |
| File name | somir.idl |
| Local-only | True |
| Ancestor interfaces | "TypedefDef Interface" on page 228 |

| | |
|---|--|
| Exceptions | "CORBA::SystemException" on page |
| Supported operations | "StructDef::members" on page 217 |
| "IDLType::type" on page 127 | |

Intended Usage

An instance of a StructDef object is used within the Interface Repository to represent an OMG IDL structure definition. An instance of a StructDef can be created using the ["create_struct" on page 93](#) operation of the Container interface.

IDL syntax

```

module CORBA
{
    struct StructMember
    {
        Identifier name;
        TypeCode type;
        IDLType type_def;
    };
    typedef sequence StructMemberSeq;
    interface StructDef: TypedefDef
    {
        attribute StructMemberSeq members;
    };
};

```

StructDef::members

| | |
|---------------------------|---|
| Overview | The members read and write operations provide for the access and update of the list of elements of an OMG IDL structure definition in the Interface Repository. |
| Original interface | " StructDef Interface" on page 216 |
| Exceptions | "CORBA::SystemException" on page |

Intended Usage

The members attribute contains a description of each structure member. The members read and write operations allow the access and update of the members attribute.

IDL Syntax

```

attribute StructMemberSeq members;

```

Read operations

Input parameters

None.

Return values

CORBA::StructMemberSeq *

The returned pointer references a sequence that is representative of the structure members. The memory is owned by the caller and can be released by invoking delete.

Write operations

Input parameters

CORBA::StructMemberSeq & members

The members parameter provides the list of structure members with which to update the StructDef.

Return values

None.

Example

```

// C++
// assume 'this_struct_def', 'pk_long_ptr', and 'pk_double_ptr'
// have already been initialized
CORBA::StructDef * this_struct_def;

```

```

CORBA::PrimitiveDef * pk_long_ptr;
CORBA::PrimitiveDef * pk_double_ptr;
// establish and initialize the StructMemberSeq . . .
CORBA::StructMemberSeq seq_update;
seq_update.length (2);
seq_update[0].name = CORBA::string_dup ("element_zero_long");
seq_update[0].type_def = CORBA::IDLType::_duplicate (pk_long_ptr);
seq_update[1].name = CORBA::string_dup ("element_one_double");
seq_update[1].type_def = CORBA::IDLType::_duplicate (pk_double_ptr);
// set the members attribute of the StructDef
this_struct_def-> members (seq_update);
// read the members attribute information from the StructDef
CORBA::StructMemberSeq * returned_members;
returned_members = this_struct_def-> members ();

```

CORBA module: SystemException Class

| | |
|--------------------------|--|
| Overview | Describes a system exception condition that has occurred. |
| File name | sys_excp.h |
| Supported methods | "SystemException::_duplicate" on page 219 "SystemException::_nil" on page 219 "systemException::completed" on page 220 "SystemException::minor" on page 220 |

Intended Usage

This class is intended to be caught in the catch clause of a try/catch block that encompasses remote method invocations or calls to ORB services. Instances of SystemException subclasses (see list below) can be thrown from implementations of IDL interfaces, to indicate some exception condition not related to the application logic (for example, unavailable memory). SystemExceptions can be thrown by any operation, regardless of the interface specification (that is, its "raises" clause in IDL).

Each SystemException contains a minor error code, to designate the subcategory of the exception, and a completion status (COMPLETED_YES, COMPLETED_NO, or COMPLETED_MAYBE) to indicate whether the object completed processing the request prior to the exception being thrown. The SystemException class provides a non-default constructor whose arguments are the minor code of the exception (of type CORBA::ULong) and the completion status (of type CORBA::CompletionStatus). When the default constructor is used, the completion status defaults to COMPLETED_NO and the minor code defaults to zero.

The subclasses of SystemException, representing specific error conditions, are defined in std_excp.h as follows:

- BAD_CONTEXT;
- BAD_INV_ORDER;
- BAD_OPERATION;
- BAD_PARAM;
- BAD_TYPECODE;
- COMM_FAILURE;
- DATA_CONVERSION;
- FREE_MEM;
- IMP_LIMIT;
- INITIALIZE;
- INTERNAL;

- INTF_REPOS;
- INVALID_TRANSACTION;
- INV_FLAG;
- INV_IDENT;
- INV_OBJREF;
- MARSHAL;
- NO_IMPLEMENT;
- NO_MEMORY;
- NO_PERMISSION;
- NO_RESOURCES;
- NO_RESPONSE;
- OBJECT_NOT_EXIST;
- OBJ_ADAPTER;
- PERSIST_STORE;
- TRANSACTION_REQUIRED;
- TRANSACTION_ROLLEDBACK;
- TRANSIENT;
- UNKNOWN;

Each subclass of SystemException has corresponding release, is_nil, _duplicate, and _nil methods, and a non-default constructor that mirrors the SystemException non-default constructor.

In the Java implementation, org.omg.CORBA.SystemException derives from java.lang.RuntimeException.

SystemException::_duplicate

| | |
|----------------|--|
| Overview | Duplicates a SystemException object. |
| Original class | "CORBA::SystemException" on page 218 |

Intended Usage

This method is intended to be used by client and server applications to duplicate a reference to a SystemException object. Both the original and the duplicate reference should subsequently be released using CORBA::release(SystemException_ptr).

IDL Syntax

```
static CORBA::SystemException_ptr _duplicate
(CORBA::SystemException_ptr p);
```

Input parameters

p

The SystemException object to be duplicated. The reference can be nil, in which case the return value will also be nil.

Return values

CORBA::SystemException_ptr

The new SystemException object reference. This value should subsequently be released using CORBA::release(SystemException_ptr).

SystemException::_nil

| | |
|-----------------------|--|
| Overview | Returns a nil CORBA::SystemException reference. |
| Original class | "CORBA::SystemException" on page 218 |

Intended Usage

This method is intended to be used by client and server applications to create a nil SystemException reference.

IDL Syntax

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

Input parameters

None.

Return values

CORBA::SystemException_ptr

A nil SystemException reference.

SystemException::completed

| | |
|-----------------------|---|
| Overview | Indicates whether an operation completed before an exception was encountered. |
| Original class | "CORBA::SystemException" on page 218 |

Intended Usage

The first completed method is intended to be used by a client or server application after catching a SystemException in a try/catch block, to determine whether the operation that threw the SystemException completed before the exception was encountered.

The second completed method is used to set the completion status of a SystemException before throwing it.

IDL Syntax

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

Input parameters

status

The completion status to store in the SystemException.

Return values

CORBA::CompletionStatus

A value indicating whether the operation that threw the SystemException completed before the exception was encountered (CORBA::COMPLETED_YES, CORBA::COMPLETED_NO, or CORBA::COMPLETED_MAYBE).

SystemException::minor

| | |
|-----------------------|--|
| Overview | Indicates the minor error code of a SystemException. |
| Original class | "CORBA::SystemException" on page 218 |

Intended Usage

The first minor method is intended to be used by a client or server application after catching a SystemException in a try/catch block, to determine the minor error code.

The second minor method is used to set the minor code of a SystemException before

throwing it.

IDL Syntax

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

Input parameters

min_id

The minor code to store in the SystemException.

Return values

CORBA::ULong

A value indicating the minor error code contained in the SystemException.

CORBA module: TypeCode Class

| | |
|--------------------------|---|
| Overview | Represents an OMG IDL type. |
| File name | typecode.h |
| Supported methods | "TypeCode::_duplicate" on page 222 |
| | "TypeCode::_nil" on page 222 |
| | "TypeCode::content_type" on page 222 |
| | "TypeCode::default_index" on page 223 |
| | "TypeCode::discriminator_type" on page 223 |
| | "TypeCode::equal" on page 224 |
| | "TypeCode::id" on page 224 |
| | "TypeCode::kind" on page 225 |
| | "TypeCode::length" on page 225 |
| | "TypeCode::member_count" on page 225 |
| | "TypeCode::member_label" on page 226 |
| | "TypeCode::member_name" on page 226 |
| | "TypeCode::member_type" on page 227 |
| | "TypeCode::name" on page 227 |
| Exceptions | BadKind An operation is not appropriate for the TypeCode kind. Bounds The index parameter is greater than or equal to the number of members constituting the type. |

Intended Usage

A TypeCode represents an OMG IDL type. A TypeCode is an integral part of the any type and is used to specify the type of the value. The Interface Repository also uses TypeCodes to store information about types declared in IDL.

A TypeCode consists of a "kind" field and zero or more parameters to fully describe the underlying data type. For example, the TypeCode describing IDL type char has kind tk_char and no parameters. The TypeCode describing the IDL type array has kind tk_array and two parameters, a TypeCode describing the type of elements in the array and a long indicating the length of the array. ["CORBA::ORB" on page 167](#) provides methods to create complex

TypeCodes (TypeCodes which have parameters). The naming convention for these methods is create_<type>_tc. For example, the method “create_array_tc” on page 171 creates a tk_array TypeCode.

Methods are provided to access the various parts of a TypeCode. Since the structure of a TypeCode varies, most methods are only applicable to certain TypeCodes. The BadKind exception is raised if a method is not applicable to the target TypeCode. Methods that deal with indexing raise the Bounds exception if the input index is greater than or equal to the number of members constituting the type. For additional information, see the “Any” on page 55 and ORB class descriptions.

TypeCode::_duplicate

| | |
|-----------------------|-------------------------------|
| Overview | Duplicates a TypeCode object. |
| Original class | “CORBA::TypeCode” on page 221 |

Intended Usage

This method is intended to be used by client and server applications to duplicate a reference to a TypeCode object. Both the original and the duplicate reference should subsequently be released using CORBA::release(TypeCode_ptr).

IDL Syntax

```
static CORBA::TypeCode_ptr _duplicate (CORBA::TypeCode_ptr p);
```

Input parameters

p

The TypeCode object to be duplicated. The reference can be nil, in which case the return value will also be nil.

Return values

CORBA::TypeCode_ptr

The new TypeCode object reference. This value should subsequently be released using CORBA::release(TypeCode_ptr).

TypeCode::_nil

| | |
|-----------------------|--|
| Overview | Returns a nil CORBA::TypeCode reference. |
| Original class | “CORBA::TypeCode” on page 221 |

Intended Usage

This method is intended to be used by client and server applications to create a nil TypeCode reference.

IDL Syntax

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

Input parameters

None.

Return values

CORBA::TypeCode_ptr

A nil TypeCode reference.

TypeCode::content_type

| | |
|-----------------|--|
| Overview | Returns the element type of a sequence or array, or the original type of an alias. |
|-----------------|--|

| | |
|-----------------------|--|
| Original class | "CORBA::TypeCode" on page 221 |
| Exceptions | "CORBA::SystemException" on page 221 |
| | "CORBA::TypeCode::BadKind" on page 221 |

Intended Usage

The `content_type` method can be invoked on sequence, array, and alias TypeCodes. For sequences and arrays, the `content_type` method returns the element type. For aliases, the `content_type` method returns the original type.

IDL Syntax

```
CORBA::TypeCode_ptr content_type() const;
```

Input parameters

None.

Return values

CORBA::TypeCode_ptr

A pointer to the element type of the sequence or array, or the original type of the alias. Ownership of the return value transfers to the caller and must be freed by calling `CORBA::release(CORBA::TypeCode_ptr)`.

TypeCode::default_index

| | |
|-----------------------|--|
| Overview | Returns the index of the default union member. |
| Original class | "CORBA::TypeCode" on page 221 |
| Exceptions | "CORBA::SystemException" on page 221 |
| | "CORBA::TypeCode::BadKind" on page 221 |

Intended Usage

The `default_index` method can only be invoked on union TypeCodes. The `default_index` method returns the index of the default member, or -1 if there is no default member.

IDL Syntax

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

Input parameters

None.

Return values

CORBA::Long

The index of the default union member.

TypeCode::discriminator_type

| | |
|-----------------------|--|
| Overview | Returns the discriminator TypeCode of a union. |
| Original class | "CORBA::TypeCode" on page 221 |
| Exceptions | "CORBA::SystemException" on page 221 |
| | "CORBA::TypeCode::BadKind" on page 221 |

Intended Usage

The `discriminator_type` method can only be invoked on union TypeCodes. The `discriminator_type` method returns the type of all non-default member labels.

IDL Syntax

```
CORBA::TypeCode_ptr discriminator_type() const;
```

Input parameters

None.

Return values

CORBA::TypeCode_ptr

A pointer to the discriminator TypeCode of the union. Ownership of the return value transfers to the caller and must be freed by calling CORBA::release(CORBA::TypeCode_ptr).

TypeCode::equal

| | |
|-----------------------|--|
| Overview | Compares two TypeCodes for equality. |
| Original class | "CORBA::TypeCode" on page 221 |
| Exceptions | "CORBA::SystemException" on page |

Intended Usage

The equal method can be invoked on any TypeCode. The equal method is used to determine whether two distinct TypeCodes describe the same underlying abstract data type. Equivalent TypeCodes produce the same results when TypeCode methods are invoked on them.

IDL Syntax

```
CORBA::Boolean equal(CORBA::TypeCode_ptr tc) const;
```

Input parameters

tc

A pointer to the TypeCode to be compared against the target TypeCode.

Return values

CORBA::Boolean

A return value of one indicates that the input TypeCode and the target TypeCode are equal. A return value of zero indicates the TypeCodes are not equal.

TypeCode::id

| | |
|--|--|
| Overview | Returns the Interface Repository identifier of an interface, structure, union, enumeration, alias, or exception. |
| Original class | "CORBA::TypeCode" on page 221 |
| Exceptions | "CORBA::SystemException" on page |
| "CORBA::TypeCode::BadKind" on page 221 | |

Intended Usage

The id method can be invoked on interface, structure, union, enumeration, alias, and exception TypeCodes. The id method returns the RepositoryId of a type.

IDL Syntax

```
const char * id() const;
```

Input parameters

None.

Return values

const char *

The Interface Repository identifier of the interface, structure, union, enumeration, alias, or exception. Ownership of the return value is maintained by the TypeCode; the return

value must not be freed by the caller.

TypeCode::kind

| | |
|----------------|---|
| Overview | Categorizes the abstract data type described by a TypeCode. |
| Original class | "CORBA::TypeCode" on page 221 |
| Exceptions | "CORBA::SystemException" on page |

Intended Usage

The kind method can be invoked on all TypeCodes. The kind method is used to classify a TypeCode into one of the categories listed in the TCKind enumeration. Based on the "kind" classification, a TypeCode may contain zero or more additional parameters to fully describe the underlying data type. See the TypeCode class description for a list of legal TypeCode kinds and parameters.

IDL Syntax

```
CORBA::TCKind kind() const;
```

Input parameters

None.

Return values

CORBA::TCKind

TCKind enumeration value.

TypeCode::length

| | |
|--|---|
| Overview | Returns the bound of a string or sequence, or the number of elements in an array. |
| Original class | "CORBA::TypeCode" on page 221 |
| Exceptions | "CORBA::SystemException" on page |
| "CORBA::TypeCode::BadKind" on page 221 | |

Intended Usage

The length method can be invoked on string, sequence, and array TypeCodes. For strings and sequences, the length method returns a bound, with zero indicating an unbounded string or sequence. For arrays, the length method returns the number of elements in an array.

IDL Syntax

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

Input parameters

None.

Return values

CORBA::ULong

The bound of the string or sequence, the number of elements in the array.

TypeCode::member_count

| | |
|----------------|---|
| Overview | Returns the number of members in a structure, union, enumeration, or exception. |
| Original class | "CORBA::TypeCode" on page 221 |

| | |
|--|--|
| Exceptions | "CORBA::SystemException" on page |
| "CORBA::TypeCode::BadKind" on page 221 | |

Intended Usage

The member_count method can be invoked on structure, union, enumeration, and exception TypeCodes. The member_count method returns the number of members constituting the type.

IDL Syntax

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

Input parameters

None.

Return values

CORBA::ULong

The number of members in the structure, union, enumeration, or exception.

TypeCode::member_label

| | |
|--|--|
| Overview | Returns the label of a union member. |
| Original class | "CORBA::TypeCode" on page 221 |
| Exceptions | "CORBA::SystemException" on page |
| "CORBA::TypeCode::BadKind" on page 221 | |
| "CORBA::TypeCode::Bounds" on page 221 | |

Intended Usage

The member_label method can only be invoked on union TypeCodes. The member_label method returns the label of the member identified by index. For the default member, the label is the zero octet.

IDL Syntax

```
CORBA::Any_ptr member_label(CORBA::ULong index) const
```

Input parameters

index

The index of the desired union member, starting at zero.

Return values

CORBA::Any_ptr

A pointer to the label of the union member. Ownership of the return value transfers to the caller and must be freed by calling CORBA::release(CORBA::TypeCode_ptr).

TypeCode::member_name

| | |
|--|--|
| Overview | Returns the simple name of a structure, union, enumeration, or exception member. |
| Original class | "CORBA::TypeCode" on page 221 |
| Exceptions | "CORBA::SystemException" on page |
| "CORBA::TypeCode::BadKind" on page 221 | |
| "CORBA::TypeCode::Bounds" on page 221 | |

Intended Usage

The `member_name` method can be invoked on structure, union, enumeration, and exception TypeCodes. The `member_name` method returns the simple name of the member identified by index.

IDL Syntax

```
const char * member_name(CORBA::ULong index) const;
```

Input parameters

index

The index of the desired member, starting at zero.

Return values

const char *

The simple name of the structure, union, enumeration, or exception member. Ownership of the return value is maintained by the TypeCode; the return value must not be freed by the caller.

TypeCode::member_type

| | |
|-----------------------|--|
| Overview | Returns the type of a structure, union, or exception member. |
| Original class | "CORBA::TypeCode" on page 221 |
| Exceptions | "CORBA::SystemException" on page 221 |
| | "CORBA::TypeCode::BadKind" on page 221 |
| | "CORBA::TypeCode::Bounds" on page 221 |

Intended Usage

The `member_type` method can be invoked on structure, union, and exception TypeCodes. The `member_type` method returns the TypeCode describing the type of the member identified by index.

IDL Syntax

```
CORBA::TypeCode_ptr member_type(CORBA::ULong index) const;
```

Input parameters

index

The index of the desired member, starting at zero.

Return values

CORBA::TypeCode_ptr

A pointer to the type of the structure, union, or exception member. Ownership of the return value transfers to the caller and must be freed by calling `CORBA::release(CORBA::TypeCode_ptr)`.

TypeCode::name

| | |
|-----------------------|--|
| Overview | Returns the simple name of an interface, structure, union, enumeration, alias, or exception. |
| Original class | "CORBA::TypeCode" on page 221 |
| Exceptions | "CORBA::SystemException" on page 221 |
| | "CORBA::TypeCode::BadKind" on page 221 |

Intended Usage

The `name` method can be invoked on object reference, structure, union, enumeration, alias, and exception TypeCodes. The `name` method returns the simple name identifying the type within its enclosing scope.

IDL Syntax

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

Input parameters

None.

Return values

const char *

The simple name of the interface, structure, union, enumeration, alias, or exception. Ownership of the return value is maintained by the TypeCode; the return value must not be freed by the caller.

CORBA module: TypedefDef Interface

| | |
|-----------------------------|---|
| Overview | An abstract interface used by the Interface Repository as a base interface to represent data types including structures, unions, enumerations, and aliases. |
| File name | somir.idl |
| Local-only | True |
| Ancestor interfaces | "Contained Interface" on page 79 "IDLType Interface" on page 127 |
| Exceptions | "CORBA::SystemException" on page |
| Supported operations | " TypedefDef::describe" on page 228 |

Intended Usage

The TypedefDef interface is not itself instantiated as a means of accessing the Interface Repository. As an ancestor to Interface Repository objects that represent OMG IDL data types, it provides a specific operation as noted below. Those Interface Repository objects that inherit (directly or indirectly) the operation defined in TypedefDef include: StructDef, UnionDef, EnumDef, and AliasDef.

IDL syntax

```
module CORBA
{
    interface TypedefDef:Contained, IDLType
    {
    };
    struct TypeDescription
    {
        Identifier name;
        RepositoryId id;
        RepositoryId defined_in;
        VersionSpec version;
        TypeCode type;
    }
}
```

TypedefDef::describe

| | |
|---------------------------|--|
| Overview | The describe operation returns a structure containing information about a CORBA::TypedefDef Interface Repository object. |
| Original interface | " TypedefDef Interface" on page 228 |
| Exceptions | "CORBA::SystemException" on page |

Intended Usage

The inherited describe operation returns a structure (CORBA::Contained::Description) that contains information about a CORBA::TypedefDef Interface Repository object. The

CORBA::Contained::Description structure has two fields: kind (CORBA::DefinitionKind data type), and value (CORBA::Any data type).

The kind of definition described by the returned structure is provided using the kind field, and the value field is a CORBA::Any that contains the description that is specific to the kind of object described. When the describe operation is invoked on a type definition (CORBA::TypedefDef) object, the kind field is representative of the specific type of CORBA::TypedefDef (either CORBA::dk_Union, CORBA::dk_Struct, CORBA::dk_Alias, or CORBA::dk_Enum). The value field contains the CORBA::TypeDescription structure.

IDL Syntax

```

struct TypeDescription
{
    Identifier name;
    RepositoryId id;
    RepositoryId defined_in;
    VersionSpec version;
    TypeCode type;
};
struct Description
{
    DefinitionKind kind;
    any value;
};
Description describe ();

```

Input parameters

None.

Return values

Description *

The returned value is pointer in a CORBA::Contained::Description structure. The memory is owned by the caller and can be removed by invoking delete.

Example

```

// C++
// assume that 'this_union' has already been initialized
CORBA::UnionDef * this_union;
// retrieve a description of the union
CORBA::UnionDef::Description * returned_description;
returned_description = this_union-> describe ();
// retrieve the type definition description from the returned
// description structure
CORBA::TypeDescription * type_description;
type_description = (CORBA::TypeDescription *) returned_description
    value.value ();

```

CORBA module: UnionDef Interface

| | |
|-----------------------------|--|
| Overview | The UnionDef interface is used within the Interface Repository to represent an OMG IDL union definition. |
| File name | somir.idl |
| Local-only | True |
| Ancestor interfaces | "TypedefDef Interface" on page 228 |
| Exceptions | "CORBA::SystemException" on page |
| Supported operations | "UnionDef::discriminator_type" on page 230 |
| | "UnionDef::discriminator_type_def" on page 230 |
| | "UnionDef::members" on page 231 |
| | "IDLType::type" on page 127 |

Intended Usage

An instance of a UnionDef object is used within the Interface Repository to represent an OMG IDL union definition. An instance of a UnionDef object can be created using the create_union operation of the Container interface.

IDL syntax

```
module CORBA
{
    struct UnionMember
    {
        Identifier name;
        anylabel;
        Typecode type;
        IDLType type_def;
    };
    typedef sequence UnionMemberSeq;
    interface UnionDef:TypeDefDef
    {
        readonlyattribute TypeCode discriminator_type;
        attribute IDLType discriminator_type_def;
        attribute UnionMemberSeq members;
    };
};
```

UnionDef::discriminator_type

| | |
|---------------------------|---|
| Overview | The discriminator_type operation returns TypeCode information representative of the discriminator of an Interface Repository UnionDef object. |
| Original interface | “UnionDef Interface” on page 229 |
| Exceptions | “CORBA::SystemException” on page |

Intended Usage

The discriminator_type attribute describes and identifies the union's discriminator type. The discriminator_type attribute can be accessed using the discriminator_type read operation. The discriminator_type attribute can only be changed by updating the discriminator_type_def attribute.

IDL Syntax

```
readonly attribute TypeCode discriminator_type;
```

Input parameters

None.

Return values

TypeCode_ptr

The returned value is a pointer to a TypeCode that represents the type of the union discriminator. The memory is owned by the caller and can be released by invoking CORBA::release.

Example

```
// C++
// assume that 'this_union' has already been initialized
CORBA::UnionDef * this_union;
// retrieve the TypeCode information that represents
// the union discriminator
CORBA::TypeCode * unions_discriminator_tc;
unions_discriminator_tc = this_union->discriminator_type();
```

UnionDef::discriminator_type_def

| | |
|---------------------------|--|
| Overview | The discriminator_type_def read and write operations allow access and update of the discriminator_type_def attribute of an Interface Repository UnionDef object. |
| Original interface | “UnionDef Interface” on page 229 |
| Exceptions | “CORBA::SystemException” on page |

Intended Usage

The discriminator_type_def attribute references an IDLType that is a type definition for the discriminator of a union. Both read and write operations are supported with parameters as defined above.

IDL Syntax

```
attribute IDLType discriminator_type_def;
```

Read operations

Input parameters

None.

Return values

CORBA::IDLType_ptr

The returned value is a pointer to a copy of the IDLType that represents the discriminator_type_def attribute. The memory is owned by the caller and can be released by invoking CORBA::release.

Write operations

Input parameters

CORBA::IDLType_ptr discriminator_type_def

The discriminator_type_def must be of a subset of the simple types⁴ or an enumeration definition (EnumDef). Setting the discriminator_type_def also updates the discriminator_type attribute.

Return values

None.

Example

```
// C++
// assume that 'this_union' and 'pk_long_ptr'
// have already been initialized
CORBA::UnionDef * this_union;
CORBA::PrimitiveDef * pk_long_ptr;
// set the discriminator_type_def to represent a CORBA::Long data type
this_union-> discriminator_type_def (pk_long_ptr);
// retrieve the discriminator_type_def information from the UnionDef
// object
CORBA::IDLType * ret_idltype_ptr;
ret_idltype_ptr = this_union-> discriminator_type_def ();
```

UnionDef::members

| | |
|---------------------------|---|
| Overview | The members read and write operations provide for the access and update of the list of elements of an OMG IDL union definition in the Interface Repository. |
| Original interface | " UnionDef Interface" on page 229 |
| Exceptions | "CORBA::SystemException" on page |

Intended Usage

The members attribute contains a description of each union member. The members read and write operations allow the access and update of the members attribute.

IDL Syntax

```
attribute UnionMemberSeq members;
```

Read operations

Input parameters

None.

Return values

CORBA::UnionMemberSeq *

The returned pointer references a sequence that is representative of the union members. The memory is owned by the caller and can be released by invoking delete.

Write operations

Input parameters

⁴ a PrimitiveDef of kind CORBA::pk_long, CORBA::pk_ulong, CORBA::pk_short, CORBA::pk_ushort, CORBA::pk_boolean, CORBA::pk_wchar, or CORBA::pk_char

CORBA::UnionMemberSeq & members

The members parameter provides the list of union members with which to update the UnionDef.

Return values

None.

Example

```
// C++
// assume 'this_union_def', 'pk_long_ptr', and 'pk_double_ptr'
// have already been initialized
CORBA::UnionDef * this_union_def;
CORBA::PrimitiveDef * pk_long_ptr;
CORBA::PrimitiveDef * pk_double_ptr;
// establish and initialize the UnionMemberSeq . . .
CORBA::UnionMemberSeq seq_update;
seq_update.length (2);
seq_update[0].name = CORBA::string_dup ("element_zero_long");
seq_update[0].label <=<= (CORBA::Long) 1;
seq_update[0].type_def = CORBA::IDLType::_duplicate (pk_long_ptr);
seq_update[1].name = CORBA::string_dup ("element_one_double");
seq_update[1].label <=<= (CORBA::Long) 2;
seq_update[1].type_def = CORBA::IDLType::_duplicate (pk_double_ptr);
// set the members attribute of the UnionDef using 'seq_update'
this_union_def-> members (seq_update);
// read the members attribute information from the UnionDef
CORBA::UnionMemberSeq * returned_members;
returned_members = this_union_def-> members ();
```

CORBA module: UnknownUserException Class

| | |
|--------------------------|--|
| Overview | Describes a generic application-specific exception condition that has occurred. |
| File name | ukn_excp.h |
| Supported methods | "UnknownUserException::_duplicate" on page 232 |
| | "UnknownUserException::_nil" on page 233 |
| | "UnknownUserException::exception" on page 233 |
| | UnknownUserException::id (inherited from "Exception" on page 121) |

Intended Usage

Request invocations made through the Dynamic Invocation Interface (DII) may result in user-defined exceptions that cannot be represented in the client program (because the exception type was not known at compile time). The CORBA::UnknownUserException class is intended to be caught in the catch clause of a try/catch block that encompasses a DII invocation.

Applications should never explicitly thrown instances of CORBA::UnknownUserException.

UnknownUserException::_duplicate

| | |
|-----------------------|---|
| Overview | Duplicates an UnknownUserException object. |
| Original class | "CORBA::UnknownUserException" on page 232 |

Intended Usage

This method is intended to be used by client and server applications to duplicate a reference to an UnknownUserException object. Both the original and the duplicate reference should subsequently be released using CORBA::release(UnknownUserException_ptr).

IDL Syntax

```
static CORBA::UnknownUserException_ptr _duplicate
```

```
(CORBA::UnknownUserException_ptr p);
```

Input parameters

p

The UnknownUserException object to be duplicated. The reference can be nil, in which case the return value will also be nil.

Return values

CORBA::UnknownUserException_ptr

The new UnknownUserException object reference. This value should subsequently be released using CORBA::release(UnknownUserException_ptr).

UnknownUserException::_nil

| | |
|-----------------------|---|
| Overview | Returns a nil CORBA::UnknownUserException reference. |
| Original class | "CORBA::UnknownUserException" on page 232 |

Intended Usage

This method is intended to be used by client and server applications to create a nil UnknownUserException reference.

IDL Syntax

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

Input parameters

None.

Return values

CORBA::UnknownUserException_ptr

A nil UnknownUserException reference.

UnknownUserException::exception

| | |
|-----------------------|---|
| Overview | Specifies the UserException contained in a CORBA::UnknownUserException. |
| Original class | "CORBA::UnknownUserException" on page 232 |

Intended Usage

This method is intended to be used by applications that catch a CORBA::UnknownUserException when attempting to invoke a method dynamically using the DII. The exception() method can be used to access the specific UserException that was thrown by the remote request.

IDL Syntax

```
CORBA::Any &exception();
```

Input parameters

None.

Return values

CORBA::Any &

An Any object whose type() indicates the type of the exception that was thrown (some subclass of CORBA::UserException) and whose value() is the exception that was thrown (an instance of some subclass of CORBA::UserException). The UnknownUserException object retains ownership of the returned Any and its contents.

CORBA module: UserException Class

| | |
|---|--|
| Overview | Describes an application-specific exception condition that has occurred. |
| File name | usr_excp.h |
| Supported methods | "UserException::_duplicate" on page 234 "UserException::_nil" on page 234 |
| UserException::id (inherited from "Exception" on page 121) | |

Intended Usage

This class is intended to be caught in the catch clause of a try/catch block that encompasses operation invocations. Typically UserException instances will actually be instances of some application-specific subclass of UserException, or ["CORBA::UnknownUserException" on page 232](#) . For each application-specific exception defined in IDL, the C++ bindings define a corresponding subclass of CORBA::UserException, which the interface implementation can throw.

In the Java implementation, org.omg.CORBA.UserException derives from java.lang.Exception.

UserException::_duplicate

| | |
|-----------------------|--|
| Overview | Duplicates a UserException object. |
| Original class | "CORBA::UserException" on page 234 |

Intended Usage

This method is intended to be used by client and server applications to duplicate a reference to a UserException object. Both the original and the duplicate reference should subsequently be released using CORBA::release(UserException_ptr).

IDL Syntax

```
static CORBA::UserException_ptr _duplicate
(CORBA::UserException_ptr p);
```

Input parameters

p

The UserException object to be duplicated. The reference can be nil, in which case the return value will also be nil.

Return values

CORBA::UserException_ptr

The new UserException object reference. This value should subsequently be released using CORBA::release(UserException_ptr).

UserException::_nil

| | |
|-----------------------|--|
| Overview | Returns a nil CORBA::UserException reference. |
| Original class | "CORBA::UserException" on page 234 |

Intended Usage

This method is intended to be used by client and server applications to create a nil UserException reference.

IDL Syntax

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

Input parameters

None.

Return values

CORBA::UserException_ptr

A nil UserException reference.

CORBA module: WstringDef Interface

| | |
|-----------------------------|---|
| Overview | The WstringDef interface is used to represent an OMG IDL bounded string of wide characters. |
| File name | somir.idl |
| Local-only | True |
| Ancestor interfaces | “IDLType Interface” on page 127 |
| Exceptions | “CORBA::SystemException” on page |
| Supported operations | “ WstringDef::bound” on page 235 |

Intended Usage

An instance of a WstringDef object is used by the Interface Repository to represent an OMG IDL bounded wide string data type. The WstringDef object is not a named Interface Repository object (it is in a group of interfaces known as Anonymous types), and as such does not reside as a named object in the Interface Repository database. An instance of a WstringDef object can be created using the [“create_wstring” on page 194](#) operation of the [“Repository interface” on page 191](#) . The WstringDef is intended to represent a string of wide characters whose kind is pk_wchar.

IDL syntax

```
module CORBA {
    interface WstringDef:IDLType {
        attribute unsigned long bound;
    };
};
```

WstringDef::bound

| | |
|---------------------------|--|
| Overview | The bound read and write operations allow the access and update of the bound attribute of a bounded string definition (CORBA::WstringDef) within the Interface Repository. |
| Original interface | “ WstringDef Interface” on page 235 |
| Exceptions | “CORBA::SystemException” on page |

Intended Usage

The bound attribute specifies the maximum number of characters in the string, and must not be zero.

IDL Syntax

```
attribute unsigned longbound;
```

Read operations

Input parameters

None.

Return values

CORBA::ULong

The returned is the current value of the bound attribute of the string definition

(CORBA::WstringDef) object.

Write operations

Input parameters

CORBA::ULong bound

The bound parameter is the new value to which the bound attribute of the CORBA::WstringDef object is set.

Return values

None.

Example

```
// C++
// assume that 'this_string' has already been initialized
CORBA::WstringDef * this_wstring;
// change the bound attribute of the string definition
CORBA::ULong new_bound = 409;
this_string-> bound (new_bound);
// obtain the bound of a string definition
CORBA::ULong returned_bound;
returned_bound = this_wstring-> bound ();
```

CosNaming in the Naming Service

The CosNaming module supports methods that allow the assigning of a name to an object (that is, creating an object-name binding in a context), then finding the object using the assigned name.

The files relating to the CosNaming module are listed in the table below.

Table: 1. Files for the CosNaming module

| | AIX | Solaris | Windows NT Visual C++ |
|------------------------|-------------------|---------------|-----------------------|
| module file name | CosNaming.idl | | |
| Java package file name | org.omg.CosNaming | | |
| C++ Header file name | CosNaming.hh | | |
| Linker files | libsomosa1.a | libsomosa1.so | somosa1m.lib |

Intended Usage

The key class in this module is the NamingContext class. Operations in this class can be used to build and manipulate a naming space. A naming space is distributed and federated. Objects in this naming space are managed objects.

Types

```
typedef string Istring;
struct NameComponent {
    Istring id;
    Istring kind;
};
typedef sequence <NameComponent> Name;
enum BindingType {nobject, ncontext};
struct Binding {
    Name binding_name;
    BindingType binding_type;
};
typedef sequence <Binding> BindingList;
```

Interfaces

For information on each of the interfaces within the CosNaming module, see the related topics.

CosNaming::BindingIterator Interface

| | |
|-----------------------------|---|
| Overview | Provides support for the Object Management Group (OMG) binding iteration. |
| File name | CosNaming.idl |
| Exceptions | "CORBA standard exceptions" on page |
| Supported operations | "BindingIterator::destroy" on page 238 |
| | "BindingIterator::next_n" on page 241 |
| | "BindingIterator::next_one" on page 242 |

Intended Usage

This class is instantiated and returned as an out parameter in the CosNaming::NamingContext::list method if the targeted naming context contains more name-object bindings than requested.

Types

```

typedef string Istring;
struct NameComponent {
    Istring id;
    Istring kind;
};
typedef sequence <NameComponent> Name;
enum BindingType {nobject, ncontext};
struct Binding {
    Name binding_name;
    BindingType binding_type;
};
typedef sequence <Binding> BindingList;

```

BindingIterator::destroy

| | |
|---------------------------|--|
| Overview | Destroys the iterator and frees allocated memory. |
| Original interface | "CosNaming::BindingIterator Interface" on page 237 |
| Exceptions | "CORBA standard exceptions" on page |

Intended Usage

This operation is intended to be used by client applications. It is not typically overridden.

IDL Syntax

```
void destroy();
```

Examples

The following examples demonstrate the usage of the CosNaming Module.

C++ Example

```

// A CosNaming usage example.
// For simplicity, error and exception checking and cleanup are omitted.
#include <CosNaming.hh>
#include <stdlib.h>
#include <fstream.h>
#define filename1 "NMUTST1.OUT"
#define filename2 "NMUTST1.OUT"
// Make the name "vehicles"
CosNaming::Name *vehiclesBindingName = new CosNaming::Name;
vehiclesBindingName->length( 1 );
(*vehiclesBindingName)[0].id = CORBA::string_dup("vehicles");
(*vehiclesBindingName)[0].kind = CORBA::string_dup("");
// Create a new naming context vehiclesNamingContext and bind it to the
// root rootNamingContext with the name "vehicles"
CosNaming::NamingContext_ptr vehiclesNamingContext =
    rootNamingContext->bind_new_context(*vehiclesBindingName);
// Make the name "vehicles.large"
CosNaming::Name *largevehiclesBindingName = new CosNaming::Name;
largevehiclesBindingName->length( 1 );
(*largevehiclesBindingName)[0].id = CORBA::string_dup("vehicles");
(*largevehiclesBindingName)[0].kind = CORBA::string_dup("large");
// create a new naming context largevehiclesNamingContext and bind
// it to the naming context vehiclesNamingContext with the name
// "vehicles.large"
CosNaming::NamingContext_ptr largevehiclesNamingContext =
    vehiclesNamingContext->bind_new_context(
        *largevehiclesBindingName);
// Make the name "vans"
CosNaming::Name *vansBindingName = new CosNaming::Name;
vansBindingName->length( 1 );
(*vansBindingName)[0].id = CORBA::string_dup("vans");
(*vansBindingName)[0].kind = CORBA::string_dup("");
// create a new naming context vansNamingContext and bind it to the
// naming context vehiclesNamingContext with the name "vans"
CosNaming::NamingContext_ptr vansNamingContext =
    vehiclesNamingContext->bind_new_context(*vansBindingName);
// Make the name "trucks"
CosNaming::Name *trucksBindingName = new CosNaming::Name;
trucksBindingName->length( 1 );
(*trucksBindingName)[0].id = CORBA::string_dup("trucks");
(*trucksBindingName)[0].kind = CORBA::string_dup("");
// create a new naming context trucksNamingContext and bind it to the
// naming context vehiclesNamingContext with the name "trucks"
CosNaming::NamingContext_ptr trucksNamingContext =
    vehiclesNamingContext->bind_new_context(*trucksBindingName);
// Make the name "chrysler"
CosNaming::Name *avehicleBindingName = new CosNaming::Name;
avehicleBindingName->length( 1 );
(*avehicleBindingName)[0].id = CORBA::string_dup("chrysler");
(*avehicleBindingName)[0].kind = CORBA::string_dup("");
// Create an object avehicleObject
ifstream strm1(filename1);
char refStr[2048];
memset(refStr, 2048, '\0');
strm1 >> refStr;
CORBA::Object_ptr avehicleObject = orb_p->string_to_object(refStr);
// Bind the object avehicleObject to the naming context
// vansNamingContext with the name "chrysler"
vansNamingContext->bind(*avehicleBindingName, avehicleObject);
// Create another object anothervehicleObject
ifstream strm2(filename2);

```

```

memset(refStr, 2048, '\0');
strm2 >> refStr;
CORBA::Object_ptr anothervehicleObject = orb_p->string_to_object(refStr);
// Rebind the object anothervehicleObject to the naming
// context vansNamingContext with the name "chrysler"
vansNamingContext->rebind(*anothervehicleObject);
// Bind the context vansNamingContext to the context
// vehiclesNamingContext with the name "vans"
largevehiclesNamingContext->bind_context(*vansBindingName,
    vansNamingContext);
// Make the name "vans.mini"
CosNaming::Name *miniVansBindingName = new CosNaming::Name;
miniVansBindingName->length( 1 );
(*miniVansBindingName)[0].id = CORBA::string_dup("vans");
(*miniVansBindingName)[0].kind = CORBA::string_dup("mini");
// Rebind the context vansNamingContext to the context
// vehiclesNamingContext with the name "vans.mini"
largevehiclesNamingContext->rebind_context(*miniVansBindingName,
    vansNamingContext);
// Unbind the object bound to vehiclesNamingContext with the
// name "vans"
largevehiclesNamingContext->unbind(*vansBindingName);
// Unbind the object bound to vehiclesNamingContext with the
// name "vans.mini"
largevehiclesNamingContext->unbind(*miniVansBindingName);
// Make the name "vehicles/vans/crysler.mini"
CosNaming::Name *aMiniVansPathName = new CosNaming::Name;
aMiniVansPathName->length( 3 );
(*aMiniVansPathName)[0].id = CORBA::string_dup("vehicles");
(*aMiniVansPathName)[0].kind = CORBA::string_dup("");
(*aMiniVansPathName)[1].id = CORBA::string_dup("vans");
(*aMiniVansPathName)[1].kind = CORBA::string_dup("");
(*aMiniVansPathName)[2].id = CORBA::string_dup("chrysler");
(*aMiniVansPathName)[2].kind = CORBA::string_dup("");
// Resolve the name from the root naming context
rootNamingContext_avehicleObject =
    rootNamingContext->resolve(*aMiniVansPathName);
// list only one binding in the naming root context rootNamingContext
// The remaining bindings can be retrieved from the binding iterator bi.
CosNaming::BindingList_var bl;
CosNaming::BindingIterator_var bi;
vehiclesNamingContext->list(1, bl, bi);
// Retrieve the next binding from the binding iterator
CosNaming::Binding_var b;
bi->next_one(b);
// Retrieve the next 2 bindings from the binding iterator
CosNaming::BindingList_var bl1;
bi->next_n(2, bl1);
// Destroy the naming context vehiclesNamingContext
largevehiclesNamingContext->destroy();
// Destroy the binding iterator bi
bi->destroy();

```

Java Example

```

// Java example
// make the name "vehicles"
org.omg.CosNaming.NameComponent[] vehiclesBindingName =
    new org.omg.CosNaming.NameComponent[1];
vehiclesBindingName[0] = new NameComponent();
vehiclesBindingName[0].id = "vehicles";
vehiclesBindingName[0].kind = "";
// Create a new naming context vehiclesNamingContext and bind it to the
// root rootNamingContext with the name "vehicles"
org.omg.CosNaming.NamingContext vehiclesNamingContext = null;
try {
    vehiclesNamingContext =
        rootNamingContext.bind_new_context(vehiclesBindingName);
}
catch (Exception e)
{
    // do error handling
}
// Make the name "vehicles.large"
org.omg.CosNaming.NameComponent[] largevehiclesBindingName =
    new org.omg.CosNaming.NameComponent[1];
largevehiclesBindingName[0] = new NameComponent();
largevehiclesBindingName[0].id = "vehicles";
largevehiclesBindingName[0].kind = "large";
// create a new naming context largevehiclesNamingContext and bind
// it to the naming context vehiclesNamingContext with the name
// "vehicles.large"
org.omg.CosNaming.NamingContext largevehiclesNamingContext = null;
try {
    largevehiclesNamingContext =
        vehiclesNamingContext.bind_new_context(largevehiclesBindingName);
}
catch (Exception e)
{
    // do error handling
}
// Make the name "vans"
org.omg.CosNaming.NameComponent[] vansBindingName =
    new org.omg.CosNaming.NameComponent[1];
vansBindingName[0] = new NameComponent();
vansBindingName[0].id = "vans";
vansBindingName[0].kind = "";
// create a new naming context vansNamingContext and bind it to the
// naming context vehiclesNamingContext with the name "vans"
org.omg.CosNaming.NamingContext vansNamingContext = null;
try {
    vansNamingContext =
        vehiclesNamingContext.bind_new_context(vansBindingName);
}
catch (Exception e)
{
    // do error handling
}

```

```

}
// Make the name "trucks"
org.omg.CosNaming.NameComponent[] trucksBindingName =
    new org.omg.CosNaming.NameComponent[1];
trucksBindingName[0] = new NameComponent();
trucksBindingName[0].id = "trucks";
trucksBindingName[0].kind = "";
// create a new naming context trucksNamingContext and bind it to the
// naming context vehiclesNamingContext with the name "trucks"
org.omg.CosNaming.NamingContext trucksNamingContext = null;
try {
    trucksNamingContext =
        vehiclesNamingContext.bind_new_context(trucksBindingName);
}
catch (Exception e)
{
    // do error handling
}
// Make the name "chrysler"
org.omg.CosNaming.NameComponent[] avehicleBindingName =
    new org.omg.CosNaming.NameComponent[1];
avehicleBindingName[0] = new NameComponent();
avehicleBindingName[0].id = "chrysler";
avehicleBindingName[0].kind = "";
// Create an object avehicleObject
BufferedReader strm1 = null;
String refStr = null;
try {
    strm1 = new BufferedReader(
        new InputStreamReader(new FileInputStream("NMUTST1.OUT")));
    refStr=strm1.readLine();
    strm1.close();
}
catch (Exception e)
{
    // do error handling
}
org.omg.CORBA.Object avehicleObject = orb.string_to_object(refStr);
// Bind the object avehicleObject to the naming context
// vansNamingContext with the name "chrysler"
try {
    vansNamingContext.bind(avehicleBindingName, avehicleObject);
}
catch (Exception e)
{
    // do error handling
}
// Create another object anothervehicleObject
BufferedReader strm2 = null;
try {
    strm2 = new BufferedReader(
        new InputStreamReader(new FileInputStream("NMUTST2.OUT")));
    refStr=strm2.readLine();
    strm2.close();
}
catch (Exception e)
{
    // do error handling
}
org.omg.CORBA.Object anothervehicleObject = orb.string_to_object(refStr);
// Rebind the object anothervehicleObject to the naming
// context vansNamingContext with the name "chrysler"
try {
    vansNamingContext.rebind(avehicleBindingName, anothervehicleObject);
}
catch (Exception e)
{
    // do error handling
}
// Bind the context vansNamingContext to the context
// vehiclesNamingContext with the name "vans"
try {
    largevehiclesNamingContext.bind_context(vansBindingName,
        vansNamingContext);
}
catch (Exception e)
{
    // do error handling
}
// Make the name "vans.mini"
org.omg.CosNaming.NameComponent[] miniVansBindingName =
    new org.omg.CosNaming.NameComponent[1];
miniVansBindingName[0] = new NameComponent();
miniVansBindingName[0].id = "vans";
miniVansBindingName[0].kind = "mini";
// Rebind the context vansNamingContext to the context
// vehiclesNamingContext with the name "vans.mini"
try {
    largevehiclesNamingContext.rebind_context(miniVansBindingName,
        vansNamingContext);
}
catch (Exception e)
{
    // do error handling
}
// Unbind the object bound to vehiclesNamingContext with the
// name "vans"
try {
    largevehiclesNamingContext.unbind(vansBindingName);
}
catch (Exception e)
{
    // do error handling
}
// Unbind the object bound to vehiclesNamingContext with the

```

```

// name "vans.mini"
try {
    largevehiclesNamingContext.unbind(miniVansBindingName);
}
catch (Exception e)
{
    // do error handling
}
// Make the name "vehicles/vans/chrysler.mini"
org.omg.CosNaming.NameComponent[] aMiniVansPathName =
    new org.omg.CosNaming.NameComponent[3];
aMiniVansPathName[0] = new NameComponent();
aMiniVansPathName[0].id = "vehicles";
aMiniVansPathName[0].kind = "";
aMiniVansPathName[1] = new NameComponent();
aMiniVansPathName[1].id = "vans";
aMiniVansPathName[1].kind = "";
aMiniVansPathName[2] = new NameComponent();
aMiniVansPathName[2].id = "chrysler";
aMiniVansPathName[2].kind = "";
// Resolve the name from the root naming context
try {
    avehicleObject = rootNamingContext.resolve(aMiniVansPathName);
}
catch (Exception e)
{
    // do error handling
}
// list only one binding in the naming root context rootNamingContext
// The remaining bindings can be retrieved from the binding iterator bi.
org.omg.CosNaming.BindingIterator bi;
org.omg.CosNaming.BindingIteratorHolder bih = new BindingIteratorHolder();
org.omg.CosNaming.Binding[] bl;
org.omg.CosNaming.BindingListHolder blh = new BindingListHolder();
vehiclesNamingContext.list(1, blh, bih);
bl = blh.value;
bi = bih.value;
// Retrieve the next binding from the binding iterator
org.omg.CosNaming.Binding b = new Binding();
org.omg.CosNaming.BindingHolder bh = new BindingHolder();
bi.next_one(bh);
b = bh.value;
// Retrieve the next 2 bindings from the binding iterator
org.omg.CosNaming.Binding[] bl1;
bi.next_n(2, blh);
bl1 = blh.value;
// Destroy the naming context vehiclesNamingContext
try {
    largevehiclesNamingContext.destroy();
}
catch (Exception e)
{
    // do error handling
}
// Destroy the binding iterator bi
try {
    bi.destroy();
}
catch (Exception e)
{
    // do error handling
}

```

BindingIterator::next_n

| | |
|---------------------------|--|
| Overview | Retrieves at most the specified number of name-object bindings. |
| Original interface | "CosNaming::BindingIterator Interface" on page 237 |
| Exceptions | "CORBA standard exceptions" on page |

Intended Usage

This operation allows clients to iterate through the bindings in the iterator. It is intended to be used by client applications. It is not typically overridden.

IDL Syntax

```

boolean next_n(
    in unsigned long how_many,
    out CosNaming::BindingList blist);

```

Input parameters

how_many

The maximum number of bindings to be returned.

blist

The returned BindingList.

Return values

TRUE

Indicates that more bindings exist.

FALSE

Indicates to the client that there are no more bindings.

Example

See the CosNaming Usage example for "[BindingIterator::destroy](#)" on page 238 .

BindingIterator::next_one

| | |
|---------------------------|--|
| Overview | Retrieves the next name-object binding. |
| Original interface | " CosNaming::BindingIterator Interface " on page 237 |
| Exceptions | " CORBA standard exceptions " on page |

Intended Usage

This operation is intended to be used by client applications. It is not typically overridden.

IDL Syntax

```
boolean next_one(out CosNaming::Binding binding);
```

Input parameters

binding

The returned Binding.

Return values

TRUE

Indicates that the next binding exists.

FALSE

Indicates that the next binding does not exist.

Example

See the CosNaming Usage example for "[BindingIterator::destroy](#)" on page 238 .

CosNaming::NamingContext Interface

| | |
|-------------------|---|
| Overview | Provides support for creating and manipulating a system naming tree, binding a name to an object in a naming context, retrieving an object from a naming context using the object name, and listing the bindings in a naming context. |
| File name | CosNaming.idl |
| Exceptions | " CORBA standard exceptions " on page and the following user exceptions: CosNaming::NamingContext::AlreadyBound Raised to indicate that an object is already bound to the name. Re-binding operations unbinds the name, then rebinds the name without raising this exception. CosNaming::NamingContext::CannotProceed{NamingContext ctx; Name rest_of_name;} Raised to indicate that the implementation has given up for some reason. The client may be able to |

| | |
|---|---|
| | <p>continue the operation using the returned naming context.</p> <p>CosNaming::NamingContext::InvalidName Raised to indicate that the name is invalid. A name with a length of zero is invalid. (This exception may be raised upon further implementation restrictions.)</p> <p>CosNaming::NamingContext::NotFound{NotFoundReason why; Name rest_of_name;} Raised to indicate that the name does not identify a binding. If a compound name is passed as an argument for the bind operation, it traverses multiple contexts. A NotFound exception is raised if any of the intermediate contexts cannot be resolved.</p> |
| Supported operations | "NamingContext::bind" on page 243 |
| "NamingContext::bind_context" on page 244 | |
| "NamingContext::bind_new_context" on page 245 | |
| "NamingContext::destroy" on page 245 | |
| "NamingContext::list" on page 246 | |
| "NamingContext::rebind" on page 247 | |
| "NamingContext::rebind_context" on page 248 | |
| "NamingContext::resolve" on page 248 | |
| "NamingContext::unbind" on page 249 | |

Intended Usage

This interface provides the operations necessary to create and manipulate a system naming tree, to bind a name to an object in a naming context, to retrieve an object from a naming context using the object name, and to list the bindings in a naming context.

Types

```
typedef string Istring;
struct NameComponent {
    Istring id;
    Istring kind;
};
typedef sequence <NameComponent> Name;
enum BindingType {nobject, ncontext};
struct Binding {
    Name binding_name;
    BindingType binding_type;
};
typedef sequence <Binding> BindingList;
```

NamingContext::bind

| | |
|---------------------------|---|
| Overview | Creates a binding in a naming context. |
| Original interface | "CosNaming::NamingContext Interface" on page 242 |
| Exceptions | <p>"CORBA standard exceptions" on page and the following "user exceptions" on page 242 :</p> <ul style="list-style-type: none"> • CosNaming::NamingContext::AlreadyBound • CosNaming::NamingContext::CannotProceed • CosNaming::NamingContext::InvalidName • CosNaming::NamingContext::NotFound |

Intended Usage

This operation is intended to be used by client applications. It is not typically overridden.

This operation creates a binding of a name to an object in a naming context. Binding a name to an object in a naming context creates a name-object association relative to the target naming context. Once an object is bound, it can be found through the resolve operation. Naming contexts that are bound using bind do not participate in name resolution when compound names are resolved - bind_context should be used to bind naming context objects.

This operation runs *resolve* to traverse a compound name. An object can be bound to multiple names in a context or across multiple contexts. Within a context, names of an object must be unique. That is, only one object can be bound to a particular name in a naming context.

IDL Syntax

```
void bind(  
    in CosNaming::Name name,  
    in Object obj);
```

Input parameters

name

The name for the binding.

obj

The object to be bound.

Return values

None.

Example

See the CosNaming Usage example for "[BindingIterator::destroy](#)" on page 238 .

NamingContext::bind_context

| | |
|---------------------------|---|
| Overview | Creates a naming context binding. |
| Original interface | "CosNaming::NamingContext Interface" on page 242 |
| Exceptions | "CORBA standard exceptions" on page and the following "user exceptions" on page 242 : <ul style="list-style-type: none">• <code>CosNaming::NamingContext::AlreadyBound</code>• <code>CosNaming::NamingContext::CannotProceed</code>• <code>CosNaming::NamingContext::InvalidName</code>• <code>CosNaming::NamingContext::NotFound</code> |

Intended Usage

This operation is intended to be used by client applications. It is not typically overridden.

This operation creates a naming context binding. Binding a name and a naming context object into a naming context creates a name-object association relative to the target naming context. Naming contexts that are bound using bind_context participate in name resolution when compound names are resolved. This operation is used to extend the naming tree by binding sub-contexts to contexts. Like an object, a naming context can be bound, using bind_context, to multiple names in a context or across multiple contexts. Within a context, the names bound to a context must be unique. That is, only one context can be bound to a particular name in a naming context.

IDL Syntax

```
void bind_context(
    in CosNaming::Name name,
    in CosNaming::NamingContext naming_context);
```

Input parameters

name

The name for the binding.

Return values

None.

Example

See the CosNaming Usage example for [“BindingIterator::destroy” on page 238](#) .

NamingContext::bind_new_context

| | |
|---------------------------|---|
| Overview | Creates a new naming context in the same server as the target naming context on which the operation was invoked and binds it to a supplied name. |
| Original interface | “CosNaming::NamingContext Interface” on page 242 |
| Exceptions | <p>“CORBA standard exceptions” on page and the following “user exceptions” on page 242 :</p> <ul style="list-style-type: none"> • CosNaming::NamingContext::AlreadyBound • CosNaming::NamingContext::CannotProceed • CosNaming::NamingContext::InvalidName • CosNaming::NamingContext::NotFound |

Intended Usage

This operation is intended to be used by client applications. It is not typically overridden.

The naming context that is created has the same implementation as the target naming context to which it is bound. This new context is created in the same process as that of the target naming context. Note that the target naming context in which the new context is bound is denoted by a name that is equivalent to the name name excluding the last name component of name.

IDL Syntax

```
CosNaming::NamingContext bind_new_context(in CosNaming::Name name);
```

Input parameters

name

The name for the naming context object binding.

Return values

CosNaming::NamingContext

Example

See the CosNaming Usage example for [“BindingIterator::destroy” on page 238](#) .

NamingContext::destroy

| | |
|---------------------------|---|
| Overview | Destroys a naming context. |
| Original interface | “CosNaming::NamingContext Interface” on page 242 |
| Exceptions | <p>“CORBA standard exceptions” on page and the following user exception:</p> <p>CosNaming::NamingContext::NotEmpty</p> |

| | |
|--|---|
| | Raised if the naming context contains any bindings. |
|--|---|

Intended Usage

This operation is intended to be used by client applications. It is not typically overridden.

This operation destroys the naming context if the context is empty. The naming context cannot contain bindings for this operation to succeed. It is the responsibility of the client to ensure that all bindings have been removed from the naming context before invoking this operation. Use the unbind operation to remove any bindings in the naming context; for more information, refer to the unbind Operation.

IDL Syntax

```
void destroy();
```

Input parameters

None.

Return values

None.

Example

See the CosNaming Usage example for "[BindingIterator::destroy](#)" on page 238 .

NamingContext::list

| | |
|--------------------|--|
| Overview | Retrieves bindings from a naming context. |
| Original interface | "CosNaming::NamingContext Interface" on page 242 |
| Exceptions | "CORBA standard exceptions" on page |

Intended Usage

This operation is intended to be used by client applications. It is not typically overridden.

This operation retrieves bindings from a naming context. At most, the operation returns a number of bindings equal to how_many in blist. If the naming context contains additional bindings, a ["BindingIterator" on page 237](#) is returned, and the calling program can iterate through the remaining bindings. If the naming context does not contain additional bindings, the BindingIterator is a NIL object reference.

The value of how_many should be less than or equal to a maximum of 1000.

The returned binding list is of type BindingList which contains a list of bindings. Each element in the list is of type Binding. Binding consists of two fields: binding_name which is the name part of the binding and binding_type which is the type of the object part of the binding. A binding type is either an object (nobject) or a naming context (ncontext).

IDL Syntax

```
void list(  
    in unsigned long how_many,  
    out CosNaming::BindingList blist,  
    out CosNaming::BindingIterator biterator);
```

Input parameters

how_many

The maximum number of bindings to install into the BindingList.

blist

The returned BindingList.

biterator

The returned BindingIteator.

Return values

None.

Example

See the CosNaming Usage example for [“BindingIteator::destroy” on page 238](#) .

NamingContext::new_context

This operation is not part of the programming model and should not be directly invoked or overridden.

NamingContext::rebind

| | |
|---------------------------|--|
| Overview | Recreates a name-object binding in a naming context even if the name is already bound in the naming context. |
| Original interface | “ CosNaming::NamingContext Interface” on page 242 |
| Exceptions | <p>“CORBA standard exceptions” on page and the following “ user exceptions” on page 242 :</p> <ul style="list-style-type: none"> • CosNaming::NamingContext::CannotProceed • CosNaming::NamingContext::InvalidName • CosNaming::NamingContext::NotFound |

Intended Usage

This operation is intended to be used by client applications. It is not typically overridden.

This operation recreates a name binding in a naming context, even if the name is already bound in the naming context. Rebinding a name and object into a naming context recreates a name-object association relative to the target naming context. Naming contexts that are bound using rebind do not participate in name resolution process when compound names are resolved.

If an object is already bound with the same name, the bound object is replaced by the passed argument obj. If the name-object binding does not exist, the rebind method behaves like the bind method.

As a developer, you can use the rebind method to replace an existing binding. You can use the rebind operation in place of the unbind and bind methods.

IDL Syntax

```
void rebind(
    in CosNaming::Name name,
    in Object obj);
```

Input parameters**name**

The name to be re-bound.

obj

The Object to be re-bound.

Return values

None.

Example

See the CosNaming Usage example for [“BindingIterator::destroy” on page 238](#) .

NamingContext::rebind_context

| | |
|---------------------------|--|
| Overview | Recreates a name-naming context binding in a target naming context, even if the name is already bound in the target naming context. |
| Original interface | “ CosNaming::NamingContext Interface” on page 242 |
| Exceptions | <p>“CORBA standard exceptions” on page and the following “ user exceptions” on page 242 :</p> <ul style="list-style-type: none"> • CosNaming::NamingContext::CannotProceed • CosNaming::NamingContext::InvalidName • CosNaming::NamingContext::NotFound |

Intended Usage

This operation is intended to be used by client applications. It is not typically overridden.

This operation recreates a binding to a naming context, even if the name is already bound in the naming context. Re-binding a name and a naming context object into a naming context recreates a name-object association relative to the target naming context. Naming contexts that are bound using rebind_context participate in name resolution when compound names are resolved.

The rebind_context operation is used to bind or replace a subcontext. If a context is already bound in a context, the bind operation raises the AlreadyBound exception. However, the rebind method replaces the bound object with the passed object.

IDL Syntax

```
void rebind_context(
    in CosNaming::Name name,
    in CosNaming::NamingContext naming_context);
```

Input parameters

name

The name to be re-bound.

naming_context

The NamingContext object to be re-bound to the name.

Return values

None.

Example

See the CosNaming Usage example for [“BindingIterator::destroy” on page 238](#) .

NamingContext::resolve

| | |
|---------------------------|--|
| Overview | Retrieves an Object bound to a name. |
| Original interface | “ CosNaming::NamingContext Interface” on page 242 |
| Exceptions | <p>“CORBA standard exceptions” on page and the following “ user exceptions” on page 242 :</p> <ul style="list-style-type: none"> • CosNaming::NamingContext::CannotProceed • CosNaming::NamingContext::InvalidName |

- `CosNaming::NamingContext::NotFound`

Intended Usage

This operation is intended to be used by client applications. It is not typically overridden.

This operation retrieves the object bound to name *name* in the target naming context. The name *name* could be a simple name. In that case *name* should match exactly the name bound to the object in the context. Or, the name *name* could be a compound name that spans multiple contexts. In this case, name resolution traverses multiple contexts. At each context traversed, the name bound to this context, in its super context, should match exactly the name component corresponding to this traversed context. The last name component of the compound name should match exactly the name bound to the object in the last traversed naming context.

The type of the returned object is not provided. Clients are responsible for "narrowing" the object to the appropriate type. Clients typically cast the returned object to a more specialized interface.

IDL Syntax

```
Object resolve(in CosNaming::Name name);
```

Input parameters

name

The name for the name-object binding.

Return values

Object

The name of the object bound to the supplied name.

Example

See the `CosNaming` Usage example for "[BindingIterator::destroy](#)" on page 238 .

NamingContext::unbind

| | |
|--------------------|--|
| Overview | Removes a name-object binding. |
| Original interface | " CosNaming::NamingContext Interface " on page 242 |
| Exceptions | <p>"CORBA standard exceptions" on page 242 and the following "user exceptions" on page 242 :</p> <ul style="list-style-type: none"> • <code>CosNaming::NamingContext::CannotProceed</code> • <code>CosNaming::NamingContext::InvalidName</code> • <code>CosNaming::NamingContext::NotFound</code> |

Intended Usage

This operation is intended to be used by client applications. It is not typically overridden.

The unbind operation removes a binding from a context. It unbinds the name *name* from the context. It is used to unregister the name *name* with the Naming Service.

This operation can also be used to unbind a naming context. If the naming context was originally bound using "[bind_context](#)" on page 244 , "[rebind_context](#)" on page 248 , "[bind](#)" on page 243 , or "[rebind](#)" on page 247 , the operation will be allowed to proceed. However, if this context was originally bound using "[bind_new_context](#)" on page 245 , then a `CORBA::PERSIST_STORE` exception will be thrown since this request would result in an orphaned name context (which is not supported). In the case of the `CORBA::PERSIST_STORE` exception, the user is required to call the "[destroy](#)" on page 245

method to unbind the name context.

IDL Syntax

```
void unbind(in CosNaming::Name name);
```

Input parameters

name

The name for the name-object binding.

Return values

None.

Example

See the CosNaming Usage example for [“BindingIterator::destroy” on page 238](#) .

CosTransactions in the Transaction Service

The CosTransactions Module is the only module in the Transaction Service.

Table: 1. Files for the CosTransactions module

| | AIX | Solaris | Windows NT Visual C++ |
|-------------------------------|-------------------------|---------------|-----------------------|
| module file name | CosTransactions.idl | | |
| Java package file name | org.omg.CosTransactions | | |
| C++ Header file name | CosTransactions.hh | | |
| Linker files | libsomosa1.a | libsomosa1.so | somosa1m.lib |

Types

Status

This type enumerates the various states through which a transaction goes during its lifetime. A special value is used to indicate that there is no transaction.

```
enum Status
{
    StatusActive,
    StatusMarkedRollback,
    StatusPrepared,
    StatusCommitted,
    StatusRolledBack,
    StatusUnknown,
    StatusNoTransaction,
    StatusPreparing,
    StatusCommitting,
    StatusRollingBack
};
```

StatusActive

The transaction has begun, has not yet been committed or rolled back, and has not been marked rollback-only.

StatusMarkedRollBack

The transaction has begun, has not yet been committed or rolled back, and has been marked rollback-only.

StatusPrepared

The transaction is "indoubt". This means the local CosTransactions::Coordinator object is waiting for information from another object to decide the outcome of the transaction. This status can be returned by a coordinator after it has prepared, or inside a Resource object's commit, rollback or commit_one_phase operation.

StatusCommitted

The transaction has been committed. This status is returned by RecoveryCoordinator::replay_completion. Note that it is not returned generally, because the objects associated with a transaction are destroyed immediately after the transaction has committed.

StatusRolledBack

The transaction has been rolled back. This status is returned in a Resource object's rollback operation, or by RecoveryCoordinator::replay_completion.

StatusUnknown

The status of the transaction is not currently known. This occurs in a subordinate server process during recovery, when the superior has not been contacted. This status is returned only by RecoveryCoordinator::replay_completion.

StatusNoTransaction

There is no current transaction. This status is returned only by Current::get_status.

Vote

This type enumerates the votes available to a Resource for the Resource::prepare operation.

```
enum Vote
{
    VoteCommit,
    VoteRollback,
    VoteReadOnly
};
```

Exceptions

HeuristicRollback

HeuristicCommit

HeuristicMixed

HeuristicHazard

Inactive

InvalidControl

NotPrepared

NoTransaction

NotSubtransaction

SubtransactionsUnavailable

Unavailable

Interfaces

For information on each of the interfaces within the CosTransactions module, see the related topics.

CosTransactions:: Control Interface

| | |
|----------------------|--|
| Overview | *** |
| File stem | CosTransactions |
| Local-only | True |
| Exceptions | "CORBA::SystemException" on page |
| Supported operations | "" on page 251 |
| *** | |

Intended Usage

IDL syntax

Control::get_coordinator

| | |
|--------------------|--|
| Overview | Returns an object that supports the "Coordinator Interface" on page 251 for the transaction represented by the Control object. |
| Original interface | "CosTransactions::Control Interface" on page 252 |
| Exceptions | TransactionRolledBack |
| Unavailable | |

Intended Usage

The Coordinator object can be used to register resources for the transaction associated with

the Control. The Unavailable exception is raised if the Control cannot provide the requested object. The TransactionRolledBack standard exception is raised if the Control object represents a transaction that has rolled back.

IDL Syntax

```
Coordinator get_coordinator()
    raises (Unavailable);
```

Input parameters

None.

Return values

Coordinator

An object that supports the [“Coordinator Interface” on page 254](#) for the transaction represented by the Control object. The caller should not free this object; the Transaction Service retains ownership of it.

Examples

The following examples demonstrate the usage of CosTransactions::Control::get_coordinator.

C++ Example

```
#include <CosTransactions.hh>
{
    CosTransactions::Current_ptr my_current;
    CosTransactions::Control_ptr control;
    CosTransactions::Coordinator_ptr coord;
    ...
    // Access the CosTransactions::Current object.
    CORBA::Object_ptr orbCurrentPtr =
        CBSeriesGlobal::orb()->resolve_initial_references("TransactionCurrent");
    my_current = CosTransactions::Current::_narrow(orbCurrentPtr);
    my_current->begin();
    control = my_current->get_control();
    coord = control->get_coordinator();
    ...
}
```

Java Example

```
import org.omg.CosTransactions.*;
{
    org.omg.CosTransactions.Current my_current;
    org.omg.CosTransactions.Control control;
    org.omg.CosTransactions.Coordinator coord;
    ...
    // Access the org.omg.CosTransactions.Current object.
    org.omg.CORBA.Object orbCurrentPtr =
        com.ibm.CBCUtil.CBSeriesGlobal.orb().resolve_initial_references(
            "TransactionCurrent");
    my_current =
        org.omg.CosTransactions.CurrentHelper.narrow(orbCurrentPtr);
    my_current.begin();
    control = my_current.get_control();
    coord = control.get_coordinator();
    ...
}
```

Control::get_terminator

| | |
|---------------------------|---|
| Overview | Returns an object that supports the “Terminator Interface” on page 276 for the transaction represented by the Control object. |
| Original interface | “CosTransactions::Control Interface” on page 252 |
| Exceptions | TransactionRolledBack |
| Unavailable | |

Intended Usage

The Terminator object can be used to rollback or commit the transaction associated with the Control. The Unavailable exception is raised if the Control cannot provide the requested object. The TransactionRolledBack standard exception is raised if the Control object represents a transaction that has rolled back.

IDL Syntax

```
Terminator get_terminator ()
    raises (Unavailable);
```

Input parameters

None.

Return values

Terminator

An object that supports the Terminator interface for the transaction represented by the Control object. It can be used to commit or roll back the transaction. The caller should not free the returned object; the Transaction Service retains ownership of it.

Examples

The following examples demonstrate the usage of CosTransactions::Control::get_terminator.

C++ Example

```
#include <CosTransactions.hh>
{
    CosTransactions::Current_ptr my_current;
    CosTransactions::Control_ptr control;
    CosTransactions::Terminator_ptr term;
    ...
    // Access the CosTransactions::Current object.
    CORBA::Object_ptr orbCurrentPtr =
        CBSeriesGlobal::orb()->resolve_initial_references("TransactionCurrent");
    my_current = CosTransactions::Current::_narrow(orbCurrentPtr);
    my_current->begin();
    control = my_current->get_control();
    term = control->get_terminator();
}
```

Java Example

```
import org.omg.CosTransactions.*;
{
    org.omg.CosTransactions.Current my_current;
    org.omg.CosTransactions.Control control;
    org.omg.CosTransactions.Terminator term;
    ...
    // Access the org.omg.CosTransactions.Current object.
    org.omg.CORBA.Object orbCurrentPtr =
        com.ibm.CBCUtil.CBSeriesGlobal.ORB().resolve_initial_references(
            "TransactionCurrent");
    my_current =
    org.omg.CosTransactions.CurrentHelper.narrow(orbCurrentPtr);
    my_current.begin();
    control = my_current.get_control();
    term = control.get_terminator();
}
```

CosTransactions::Coordinator Interface

| | |
|-----------------------------|---|
| Overview | Provides common operations for top-level transactions. |
| File stem | CosTransactions |
| Exceptions | Inactive |
| SubtransactionsUnavailable | |
| Supported operations | "Coordinator::get_parent_status" on page 255 "Coordinator::get_status" on page 256 |

| |
|---|
| "Coordinator::get_top_level_status" on page 256 |
| "Coordinator::get_transaction_name" on page 257 |
| "Coordinator::hash_top_level_transaction" on page 258 |
| "Coordinator::hash_transaction" on page 258 |
| "Coordinator::is_ancestor_transaction" on page 259 |
| "Coordinator::is_descendant_transaction" on page 260 |
| "Coordinator::is_related_transaction" on page 260 |
| "Coordinator::is_same_transaction" on page 261 |
| "Coordinator::is_top_level_transaction" on page 262 |
| "Coordinator::rollback_only" on page 263 |

Intended Usage

The Coordinator interface provides common operations for top-level transactions. Participants in a transaction are typically either recoverable objects, or agents of recoverable objects, such as subordinate coordinators. An object supporting the Coordinator interface is implicitly associated with one transaction only.

IDL syntax

```

interface Coordinator
{
    Status get_status();
    Status get_parent_status();
    Status get_top_level_status();
    boolean is_same_transaction(in Coordinator coord);
    boolean is_related_transaction(in Coordinator coord);
    boolean is_ancestor_transaction(in Coordinator coord);
    boolean is_descendant_transaction(in Coordinator coord);
    boolean is_top_level_transaction();
    unsigned long hash_transaction(in unsigned long maximum);
    unsigned long hash_top_level_tran(in unsigned long maximum);
    RecoveryCoordinator register_resource(in Resource res)
        raises(Inactive);
    void register_synchronization (in Synchronization sync)
        raises(Inactive);
    void register_subtran_aware(in SubtransactionalAwareResource
res)
        raises(Inactive, SubtransactionsUnavailable);
    void rollback_only()
        raises(Inactive);
    string get_transaction_name();
    Control create_subtransaction()
        raises(SubtransactionsUnavailable Inactive);
    CostTSInteroperation::PropagationContext get_txcontext ()
        raises(Unavailable);
};

```

Coordinator::get_parent_status

| | |
|---------------------------|--|
| Overview | Returns to a caller the status of the parent transaction of the transaction associated with the target object. |
| Original interface | "CosTransactions::Coordinator Interface" on page 254 |

Intended Usage

If the transaction associated with the target object is a top-level transaction, this operation is equivalent to the ["Coordinator::get_status" on page 256](#) operation.

If the transaction is not a top-level transaction, this operation returns the status of the parent of the transaction associated with the target object.

IDL Syntax

```

Status get_parent_status();

```

Input parameters

None.

Return values

Status

The status of the parent transaction of the transaction associated with the target object.

Examples

The following examples demonstrate the usage of `CosTransactions::Coordinator::get_parent_status`.

C++ Example

```
CosTransactions::Coordinator *coord;  
CosTransactions::Status *status;  
status = coord->get_parent_status();
```

Java Example

```
org.omg.CosTransactions.Coordinator coord;  
org.omg.CosTransactions.Status status;  
status = coord.get_parent_status();
```

Coordinator::get_status

| | |
|--------------------|--|
| Overview | Returns the status of the transaction. |
| Original interface | "CosTransactions::Coordinator Interface" on page 254 |

Intended Usage

The operation returns to a caller the status of the transaction associated with the target object.

IDL Syntax

```
Status get_status();
```

Input parameters

None.

Return values

Status

The status of the transaction associated with the target object.

Examples

The following examples demonstrate the usage of `CosTransactions::Coordinator::get_status`.

C++ Example

```
CosTransactions::Coordinator *coord;  
CosTransactions::Status *status;  
status = coord->get_status();
```

Java Example

```
org.omg.CosTransactions.Coordinator coord;  
org.omg.CosTransactions.Status status;  
status = coord.get_status();
```

Coordinator::get_top_level_status

| | |
|----------|--|
| Overview | Returns to a caller the status of the top-level ancestor of the transaction associated with the target object. |
|----------|--|

| | |
|--------------------|---|
| Original interface | " CosTransactions::Coordinator Interface" on page 254 |
|--------------------|---|

Intended Usage

If the transaction is a top-level transaction, this operation is equivalent to the ["Coordinator::get_status" on page 256](#) Operation. If it is not a top level transaction, this operation gets the status of the top-level ancestor.

IDL Syntax

```
Status get_top_level_status();
```

Input parameters

None.

Return values

Status

The status of the top-level ancestor of the transaction associated with the target object.

Examples

The following examples demonstrate the usage of CosTransactions::Coordinator::get_top_level_status.

C++ Example

```
CosTransactions::Coordinator *coord;
CosTransactions::Status *status;
status = coord->get_top_level_status();
```

Java Example

```
org.omg.CosTransactions.Coordinator coord;
org.omg.CosTransactions.Status status;
status = coord.get_top_level_status();
```

Coordinator::get_transaction_name

| | |
|--------------------|--|
| Overview | Supports debugging by returning a string describing the transaction associated with the target object. |
| Original interface | " CosTransactions::Coordinator Interface" on page 254 |

Intended Usage

Returns a printable string describing the transaction. If there is no transaction associated with the current thread, an empty string is returned.

IDL Syntax

```
string get_transaction_name();
```

Input parameters

None.

Return values

string

A string describing the transaction associated with the target object.

Examples

The following examples demonstrate the usage of CosTransactions::Coordinator::get_transaction_name.

C++ Example

```
CosTransactions::Coordinator *coord;
string name;
name = coord->get_transaction_name();
cout << "Transaction name is " << name << endl;
```

Java Example

```
org.omg.CosTransactions.Coordinator coord;
string name;
name = coord.get_transaction_name();
System.out.println ("Transaction name is " + name);
```

Coordinator::get_txcontext

This operation is not part of the programming model and should not be directly invoked or overridden.

Coordinator::hash_top_level_transaction

| | |
|---------------------------|--|
| Overview | Returns a hash value based on the top-level ancestor of the transaction associated with the target object. |
| Original interface | "CosTransactions::Coordinator Interface" on page 254 |

Intended Usage

Each transaction has a single hash value. Hash values for transactions should be uniformly distributed. This operation is equivalent to the ["Coordinator::hash_transaction" on page 258](#) Operation when the transaction associated with the target object is a top-level transaction.

IDL Syntax

```
unsigned long hash_top_level_tran();
```

Input parameters

None.

Return values

unsigned long

A hash value based on the top-level ancestor of the transaction associated with the target object.

Examples

The following examples demonstrate the usage of `CosTransactions::Coordinator::hash_top_level_transaction`.

C++ Example

```
CosTransactions::Coordinator *coord;
unsigned long hashval;
hashval = coord->hash_top_level_tran();
```

Java Example

```
org.omg.CosTransactions.Coordinator coord;
int hashval;
hashval = coord.hash_top_level_tran();
```

Coordinator::hash_transaction

| | |
|---------------------------|--|
| Overview | Returns a hash value based on the transaction associated with the target object. |
| Original interface | "CosTransactions::Coordinator Interface" on page 254 |

Intended Usage

Each transaction has a single hash value although multiple transactions may share the same value. Hash values for transactions should be uniformly distributed.

IDL Syntax

```
unsigned long hash_transaction();
```

Input parameters

None.

Return values
unsigned long

A hash value based on the transaction associated with the target object.

Examples

The following examples demonstrate the usage of CosTransactions::Coordinator::hash_transaction.

C++ Example

```
CosTransactions::Coordinator *coord;  
unsigned long hashval;  
hashval = coord->hash_transaction();
```

Java Example

```
org.omg.CosTransactions.Coordinator coord;  
int hashval;  
hashval = coord.hash_transaction();
```

Coordinator::is_ancestor_transaction

| | |
|---------------------------|--|
| Overview | Determines whether the transaction associated with the target object is an ancestor of the transaction associated with the parameter object. |
| Original interface | " CosTransactions::Coordinator Interface" on page 254 |

Intended Usage

A transaction is an ancestor of another transaction if, and only if, the transactions are the same, or the first is an ancestor of the parent of the second.

IDL Syntax

```
boolean is_ancestor_transaction(in Coordinator tc);
```

Input parameters

tc

A pointer to the Coordinator object for a transaction.

Return values

TRUE

The transaction associated with the target object is an ancestor of the transaction associated with the parameter object.

FALSE

The transaction associated with the target object is not an ancestor of the transaction associated with the parameter object.

Examples

The following examples demonstrate the usage of CosTransactions::Coordinator::is_ancestor_transaction.

C++ Example

```
CosTransactions::Coordinator *c1, *c2;  
if( c1->is_ancestor_transaction(c2) )  
{  
    cout << "c1 is an ancestor of c2" << endl;  
}  
else  
{  
    cout << "c1 is not an ancestor of c2" << endl;  
}
```

Java Example

```

org.omg.CosTransactions.Coordinator c1, c2;
if( c1.is_ancestor_transaction(c2) )
{
    System.out.println ("c1 is an ancestor of c2");
}
else
{
    System.out.println ("c1 is not an ancestor of c2");
}

```

Coordinator::is_descendant_transaction

| | |
|---------------------------|---|
| Overview | Determines whether the transaction associated with the target object is a descendant of the transaction associated with the parameter object. |
| Original interface | "CosTransactions::Coordinator Interface" on page 254 |

Intended Usage

A transaction is an descendant of another transaction if, and only if, the second transaction is an ancestor of the first. For an definition of ancestors of transactions, see the ["Coordinator::is_ancestor_transaction" on page 259](#) Operation.

IDL Syntax

```
boolean is_descendant_transaction(in Coordinator tc);
```

Input parameters

tc

A pointer to the Coordinator object for a transaction.

Return values

TRUE

The transaction associated with the target object is a descendant of the transaction associated with the parameter object.

FALSE

The transaction associated with the target object is not a descendant of the transaction associated with the parameter object.

Examples

The following examples demonstrate the usage of `CosTransactions::Coordinator::is_descendant_transaction`.

C++ Example

```

CosTransactions::Coordinator *c1, *c2;
if( c1->is_descendant_transaction(c2) )
{
    cout << "c1 is a descendant of c2" << endl;
}
else
{
    cout << "c1 is not a descendant of c2" << endl;
}

```

Java Example

```

org.omg.CosTransactions.Coordinator c1, c2;
if( c1.is_descendant_transaction(c2) )
{
    System.out.println ("c1 is a descendant of c2");
}
else
{
    System.out.println ("c1 is not a descendant of c2");
}

```

Coordinator::is_related_transaction

| | |
|---------------------------|--|
| Overview | Determines whether the transaction associated with the target object is related to the transaction associated with the parameter object. |
| Original interface | " CosTransactions::Coordinator Interface" on page 254 |

Intended Usage

A transaction is related to another transaction if, and only if, they share a common ancestor transaction.

IDL Syntax

```
boolean is_related_transaction(in Coordinator tc);
```

Input parameters

tc

A pointer to the Coordinator object for a transaction.

Return values

TRUE

The transaction associated with the target object is related to the transaction associated with the parameter object.

FALSE

The transaction associated with the target object is not related to the transaction associated with the parameter object.

Examples

The following examples demonstrate the usage of `CosTransactions::Coordinator::is_related_transaction`.

C++ Example

```
CosTransactions::Coordinator *c1, *c2;
if( c1->is_related_transaction(c2) )
{
    cout << "c1 is related to c2" << endl;
}
else
{
    cout << "c1 is not related to c2" << endl;
}
```

Java Example

```
org.omg.CosTransactions.Coordinator c1, c2;
if( c1.is_related_transaction(c2) )
{
    System.out.println ("c1 is related to c2");
}
else
{
    System.out.println ("c1 is not related to c2");
}
```

Coordinator::is_same_transaction

| | |
|---------------------------|---|
| Overview | Determines whether the target object and the parameter object both refer to the same transaction. |
| Original interface | " CosTransactions::Coordinator Interface" on page 254 |

Intended Usage

Determines whether the target object and the parameter object both refer to the same transaction.

IDL Syntax

```
boolean is_same_transaction(in Coordinator tc);
```

Input parameters

tc

A pointer to the Coordinator object for a transaction.

Return values

TRUE

The target object and the parameter object both refer to the same transaction.

FALSE

The target object and the parameter object do not both refer to the same transaction.

Examples

The following examples demonstrate the usage of `CosTransactions::Coordinator::is_same_transaction`.

C++ Example

```
/* C++ example */
CosTransactions::Coordinator *c1, *c2;
if( c1->is_same_transaction(c2) )
{
    cout << "c1 represents the same transaction as c2" << endl;
}
else
{
    cout << "c1 does not represent the same transaction as c2" <<
endl;
}
```

Java Example

```
org.omg.CosTransactions.Coordinator c1, c2;
if( c1.is_same_transaction(c2) )
{
    System.out.println ("c1 represents the same transaction as c2");
}
else
{
    System.out.println ("c1 does not represent the same transaction
as c2");
}
```

Coordinator::is_top_level_transaction

| | |
|---------------------------|--|
| Overview | Determines whether the transaction associated with the target object is a top-level transaction. |
| Original interface | "CosTransactions::Coordinator Interface" on page 254 |

Intended Usage

Determines whether the transaction associated with the target object is a top-level transaction. A transaction is a top-level transaction if it has no parent.

IDL Syntax

```
boolean is_top_level_transaction();
```

Input parameters

tc

A pointer to a Coordinator interface transaction.

Return values

TRUE

The transaction associated with the target object is a top-level transaction.

FALSE

The transaction associated with the target object is not a top-level transaction.

Examples

The following examples demonstrate the usage of `CosTransactions::Coordinator::is_top_level_transaction`.

C++ Example

```
CosTransactions::Coordinator *coord;
if( coord->is_top_level_transaction() )
{
    cout << "Coordinator represents a top-level transaction" <<
endl;
}
else
{
    cout << "Coordinator represents a subtransaction" << endl;
}
```

Java Example

```
org.omg.CosTransactions.Coordinator coord;
if( coord.is_top_level_transaction() )
{
    System.out.println ("Coordinator represents a top-level
transaction");
}
else
{
    System.out.println ("Coordinator represents a subtransaction");
}
```

Coordinator::register_resource

This operation is not part of the programming model and should not be directly invoked or overridden.

Coordinator::register_subtran_aware

This operation is not part of the programming model and should not be directly invoked or overridden.

Coordinator::register_synchronization

This operation is not part of the programming model and should not be directly invoked or overridden.

Coordinator::rollback_only

| | |
|--------------------|--|
| Overview | Modifies the transaction associated with the target object so that it cannot be committed, but only rolled back. |
| Original interface | " CosTransactions::Coordinator Interface" on page 254 |
| Exceptions | Inactive |

Intended Usage

Modifies the transaction associated with the target object so that it cannot be committed, but only rolled back. If the transaction is completing, the Inactive exception is raised.

IDL Syntax

```
void rollback_only()
raises (Inactive);
```

Input parameters

None.

Return values

None.

Examples

The following examples demonstrate the usage of `CosTransactions::Coordinator::rollback_only`.

C++ Example

```
CosTransactions::Coordinator *coord;  
coord->rollback_only();
```

Java Example

```
org.omg.CosTransactions.Coordinator coord;  
coord.rollback_only();
```

Current::begin

| | |
|---|--|
| Overview | Creates a new transaction. |
| Original interface | "CosTransactions::Current Interface" on page |
| Exceptions | SubtransactionsUnavailable |
| "INITIALIZE standard exception" on page | |

Intended Usage

The transaction context of the current thread is modified so that the thread is associated with the new transaction.

The `SubtransactionsUnavailable` exception is raised if the current thread already has an associated transaction and the transaction framework does not support subtransactions, or the current transaction is completing.

The `INITIALIZE` standard exception is raised if `begin` is being used for the first time and the Transaction Service cannot be initialized.

IDL Syntax

```
void begin() raises(SubtransactionsUnavailable);
```

Input parameters

None.

Return values

None.

Examples

The following examples demonstrate the usage of `CosTransactions::Current::begin`.

C++ Example

```
#include <CosTransactions.hh> // CosTransactions module  
...  
// Access the CosTransactions::Current object.  
CORBA::Object_ptr orbCurrentPtr =  
    CBSeriesGlobal::orb()->resolve_initial_references("TransactionCurrent");  
CosTransactions::Current_ptr current =  
    CosTransactions::Current::_narrow(orbCurrentPtr);  
...  
// Invoke the begin operation on the CosTransactions::Current  
object.  
current->begin();  
...  
...
```

Java Example

```
import org.omg.CosTransactions.*; // CosTransactions module  
...  
// Access the org.omg.CosTransactions.Current object.  
org.omg.CORBA.Object orbCurrentPtr =
```

```

com.ibm.CBCUtil.CBSeriesGlobal.orb().resolve_initial_references(
    "TransactionCurrent");
org.omg.CosTransactions.Current current =
    org.omg.CosTransactions.CurrentHelper.narrow(orbCurrentPtr);
...
// Invoke the begin operation on the org.omg.CosTransactions.Current
object.
current.begin();
...

```

Current::commit

| | |
|--|--|
| Overview | Completes the current transaction. |
| Original interface | "CosTransactions::Current Interface" on page |
| Exceptions | HeuristicHazard |
| HeuristicMixed | |
| NoTransaction | |
| "NO_PERMISSION standard exception" on page | |

Intended Usage

If there is no current transaction, the NoTransaction exception is raised. If the current thread does not have permission to commit the transaction, the standard NO_PERMISSION exception is raised. (The commit operation is restricted to the transaction originator in some circumstances.)

The effect of this operation is equivalent to performing the ["Terminator::commit" on page 278](#) Operation in the corresponding Terminator Interface.

The current thread transaction is modified as follows: If the transaction has been begun by a thread (using begin) in the same execution environment, the thread's transaction context is restored to its state prior to the begin request. Otherwise, the thread's transaction context is set to NULL.

If transactions are rolling back for no apparent reason you may be trying to invoke the commit operation on an object with an active exception.

IDL Syntax

```

void commit(in boolean report_heuristics)
    raises (NoTransaction,HeuristicMixed,HeuristicHazard);

```

Input parameters

report_heuristics

Flag indicating whether heuristic reporting is required.

Return values

None.

Examples

The following examples demonstrate the usage of CosTransactions::Current::commit.

C++ Example

```

#include <CosTransactions.hh> // CosTransactions module
...
::CORBA::Boolean rollback_required = FALSE;
...
//Access the CosTransactions::Current object.
CORBA::Object_ptr orbCurrentPtr =
    CBSeriesGlobal::orb()->resolve_initial_references("TransactionCurrent");
CosTransactions::Current_ptr current =
    CosTransactions::Current::_narrow(orbCurrentPtr);
// Invoke the begin operation on the CosTransactions::Current
object.

```

```

current->begin();
// Perform work for the transaction and set rollback_required to
TRUE if
// an error is detected.
...
// Invoke commit or rollback depending on whether rollback_required
is
// set. This must be called within a try...catch structure as the
// transaction service may raise an exception if an error occurs.
try
{
    if (rollback_required == TRUE)
    {
        current->rollback();
    }
    else // commit required
    {
        current->commit(/* report_heuristics = */ TRUE);
    }
}
catch (CORBA::TRANSACTION_ROLLEDBACK &exc)
{
    // The application called commit, but the transaction service
    rolled
    // the transaction back because an error was detected.
    ...
}
catch (CosTransactions::HeuristicMixed &exc)
{
    // The transaction service has reported that some or all of
    the
    // resource objects have made a heuristic decision. This
    has
    // resulted in heuristic damage.
    ...
}
catch (CosTransactions::HeuristicHazard &exc)
{
    // The transaction service has reported that not all of the
    determining the
    // resource objects could participate properly in
    // outcome of the transaction. There is a possibility of
    // heuristic damage.
    ...
}
catch (CORBA::UserException &exc)
{
    // Another type of user exception has occurred.
    ...
}
catch (CORBA::SystemException &exc)
{
    // The application called commit, but the transaction
    service
    // rolled the transaction back because an error was
    detected.
    ...
}
catch (...)
{
    // A general exception has occurred.
    ...
}
...

```

Java Example

```

import org.omg.CosTransactions.*; // CosTransactions module
...
org.omg.CORBA.Boolean rollback_required = FALSE;
...
//Access the org.omg.CosTransactions.Current object.
org.omg.CORBA.Object orbCurrentPtr =
    com.ibm.CBCUtil.CBSeriesGlobal.ORB().resolve_initial_references(
        "TransactionCurrent");
org.omg.CosTransactions.Current current =
    org.omg.CosTransactions.CurrentHelper.narrow(orbCurrentPtr);
// Invoke the begin operation on the org.omg.CosTransactions.Current
object.
current.begin();
// Perform work for the transaction and set rollback_required to
TRUE if
// an error is detected.
...

```

```

// Invoke commit or rollback depending on whether rollback_required
is // set. This must be called within a try...catch structure as the
// transaction service may raise an exception if an error occurs.
try
{
    if (rollback_required == TRUE)
    {
        current.rollback();
    }
    else // commit required
    {
        current.commit(/* report_heuristics = */ TRUE);
    }
}
catch (org.omg.CORBA.TRANSACTION_ROLLEDBACK exc)
{
    // The application called commit, but the transaction service
    rolled // the transaction back because an error was detected.
    ...
}
catch (org.omg.CosTransactions.HeuristicMixed exc)
{
    // The transaction service has reported that some or all of
    the // resource objects have made a heuristic decision. This
    has // resulted in heuristic damage.
    ...
}
catch (org.omg.CosTransactions.HeuristicHazard exc)
{
    // The transaction service has reported that not all of the
    determining the // resource objects could participate properly in
    // the outcome of the transaction. There is a possibility of
    // heuristic damage.
    ...
}
catch (org.omg.CORBA.UserException exc)
{
    // Another type of user exception has occurred.
    ...
}
catch (org.omg.CORBA.SystemException exc)
{
    // The application called commit, but the transaction
    service // rolled the transaction back because an error was
    detected.
    ...
}
catch (Exception exc)
{
    // A general exception has occurred.
    ...
}
...

```

Current::get_control

| | |
|---------------------------|--|
| Overview | Returns an object representing the transaction context currently associated with the current thread. |
| Original interface | "CosTransactions::Current Interface" on page |

Intended Usage

If the current thread is not associated with a transaction, a NULL object reference is returned.

The Control object returned can be given to the ["Current::resume" on page 270](#) operation to reestablish this transaction context in the same thread or a different thread.

IDL Syntax

```
Control get_control();
```

Input parameters

None.

Return values

Control

Represents the transaction context currently associated with the current thread. The caller should not free the returned object; the Transaction Service retains ownership of it.

Examples

The following examples demonstrate the usage of `CosTransactions::Current::get_control`.

C++ Example

```
#include <CosTransactions.hh>
{
    CosTransactions::Current_ptr my_current;
    CosTransactions::Control_ptr control;

    /** Access the CosTransactions::Current object.
    CORBA::Object_ptr orbCurrentPtr =
        CBSeriesGlobal::orb()->resolve_initial_references("TransactionCurrent");
    my_current = CosTransactions::Current::_narrow(orbCurrentPtr);
    control = my_current->get_control();
    ...
}
```

Java Example

```
import org.omg.CosTransactions.*;
{
    org.omg.CosTransactions.Current my_current;
    org.omg.CosTransactions.Control control;

    /** Access the org.omg.CosTransactions.Current object.
    org.omg.CORBA.Object orbCurrentPtr =
        com.ibm.CBCUtil.CBSeriesGlobal.ORB().resolve_initial_references(
            "TransactionCurrent");
    my_current =
    org.omg.CosTransactions.CurrentHelper.narrow(orbCurrentPtr);
    control = my_current.get_control();
    ...
}
```

Current::get_status

| | |
|--------------------|--|
| Overview | Determines the status of the current transaction. |
| Original interface | "CosTransactions::Current Interface" on page |

Intended Usage

If there is no transaction associated with the current thread, the `StatusNoTransaction` value is returned.

The effect of this request is equivalent to performing the ["get_status" on page 256](#) Operation in the corresponding ["Coordinator" on page 254](#) Interface.

IDL Syntax

```
Status get_status();
```

Input parameters

None.

Return values

Status

The status of the current transaction.

Examples

The following examples demonstrate the usage of `CosTransactions::Current::get_status`.

C++ Example

```
#include <CosTransactions.hh>
{
    CosTransactions::Current_ptr my_current;
    CosTransactions::Status_ptr status;
    ...
    // Access the CosTransactions::Current object.
    CORBA::Object_ptr orbCurrentPtr =
        CBSeriesGlobal::orb()->resolve_initial_references("TransactionCurrent");
    my_current = CosTransactions::Current::_narrow(orbCurrentPtr);
    status = my_current->get_status();
    ...
}
```

Java Example

```
import org.omg.CosTransactions.*;
{
    org.omg.CosTransactions.Current my_current;
    org.omg.CosTransactions.Status status;
    ...
    // Access the org.omg.CosTransactions.Current object.
    org.omg.CORBA.Object orbCurrentPtr =
        com.ibm.CBCUtil.CBSeriesGlobal.orb().resolve_initial_references(
            "TransactionCurrent");
    my_current =
    org.omg.CosTransactions.CurrentHelper.narrow(orbCurrentPtr);
    status = my_current.get_status();
    ...
}
```

Current::get_transaction_name

| | |
|---------------------------|--|
| Overview | Supports debugging by returning a string describing the transaction. |
| Original interface | "CosTransactions::Current Interface" on page |

Intended Usage

If there is no transaction associated with the current thread, an empty string is returned.

The effect of this request is equivalent to performing the ["get_transaction_name" on page 257](#) Operation in the corresponding ["Coordinator" on page 254](#) Interface.

IDL Syntax

```
string get_transaction_name();
```

Input parameters

None.

Return values

string

A printable string describing the transaction.

Examples

The following examples demonstrate the usage of `CosTransactions::Current::get_transaction_name`.

C++ Example

```
#include <CosTransactions.hh>
{
    CosTransactions::Current_ptr my_current;
    string name;
    ...
    // Access the CosTransactions::Current object.
    CORBA::Object_ptr orbCurrentPtr =
        CBSeriesGlobal::orb()->resolve_initial_references("TransactionCurrent");
    my_current = CosTransactions::Current::_narrow(orbCurrentPtr);
    name = my_current->get_transaction_name();
}
```

```

    cout << "Current transaction name is " << name << endl;
    ...
}

```

Java Example

```

import org.omg.CosTransactions.*;

{
    org.omg.CosTransactions.Current my_current;
    String name;
    ...
    // Access the org.omg.CosTransactions.Current object.
    org.omg.CORBA.Object orbCurrentPtr =
        com.ibm.CBCUtil.CBSeriesGlobal.orb().resolve_initial_references(
            "TransactionCurrent");
    my_current =
org.omg.CosTransactions.CurrentHelper.narrow(orbCurrentPtr);
    name = my_current.get_transaction_name();
    System.out.println ("Current transaction name is " + name);
    ...
}

```

Current::resume

| | |
|---|--|
| Overview | Associates the current thread with the supplied transaction context (in place on any previously associated transaction context). |
| Original interface | " CosTransactions::Current Interface" on page |
| Exceptions | InvalidControl |
| "INITIALIZE standard exception" on page | |

Intended Usage

If the parameter is not valid in the current execution context (that is, it is an object reference with an invalid value), the current thread is associated with no transaction.

The INITIALIZE standard exception is raised if resume is being used for the first time and the Transaction Service cannot be initialized.

IDL Syntax

```

void resume(in Control which)
    raises(InvalidControl);

```

Input parameters which

The Control object representing the transaction context.

Return values

None.

Examples

The following examples demonstrate the usage of CosTransactions::Current::resume.

C++ Example

```

#include <CosTransactions.hh> // CosTransactions module
...
CosTransactions::Control_ptr control = control_parameter;
...
// Access the CosTransactions::Current object.
CORBA::Object_ptr orbCurrentPtr =
    CBSeriesGlobal::orb()->resolve_initial_references("TransactionCurrent");
CosTransactions::Current_ptr current =
    CosTransactions::Current::_narrow(orbCurrentPtr);
...
// Resume the association between the transaction and the thread.
try
{
    current->resume(control);
}

```

```

    catch(CosTransactions::InvalidControl)
    {
        cout << "Error: control object passed to resume was invalid" <<
endl;
    }
}

```

Java Example

```

import org.omg.CosTransactions.*; // CosTransactions module
...
org.omg.CosTransactions.Control control = control_parameter;
...
//Access the org.omg.CosTransactions.Current object.
org.omg.CORBA.Object orbCurrentPtr =
com.ibm.CBCUtil.CBSeriesGlobal.ORB().resolve_initial_references(
"TransactionCurrent");
org.omg.CosTransactions.Current current =
org.omg.CosTransactions.CurrentHelper.narrow(orbCurrentPtr);
...
// Resume the association between the transaction and the thread.
try
{
    current.resume(control);
}
catch(org.omg.CosTransactions.InvalidControl exc)
{
    System.out.println ("Error: control object passed to resume was
invalid");
}

```

Current::rollback

| | |
|--|--|
| Overview | Rolls back the transaction associated with the current thread. |
| Original interface | "CosTransactions::Current Interface" on page |
| Exceptions | NoTransaction |
| "NO_PERMISSION standard exception" on page | |

Intended Usage

If there is no transaction associated with the current thread, the NoTransaction exception is raised. If the current thread does not have permission to rollback the transaction, the standard NO_PERMISSION exception is raised.

The effect of this request is equivalent to performing the ["rollback" on page 270](#) operation in the corresponding ["Terminator" on page 271](#) interface.

The current thread transaction is modified as follows: If the transaction has been begun by a thread (using begin) in the same execution environment, the thread's transaction context is restored to its state prior to the begin request. Otherwise, the thread's transaction context is set to NULL.

IDL Syntax

```
void rollback() raises(NoTransaction);
```

Input parameters

None.

Return values

Examples

The following examples demonstrate the usage of CosTransactions::Current::rollback.

C++ Example

```

#include <CosTransactions.hh> // CosTransactions module
...
::CORBA::Boolean rollback_required = FALSE;
...

```

```

//Access the CosTransactions::Current object.
CORBA::Object_ptr orbCurrentPtr =
    CBSeriesGlobal::orb()->resolve_initial_references("TransactionCurrent");
CosTransactions::Current_ptr current =
    CosTransactions::Current::_narrow(orbCurrentPtr);
// Invoke the begin operation on the CosTransactions::Current
object.
current->begin();
// Perform work for the transaction and set rollback_required to
TRUE if
// an error is detected.
...
// Invoke commit or rollback depending on whether rollback_required
is
// set. This must be called within a try...catch structure as the
// transaction service may raise an exception if an error occurs.
try
{
    if (rollback_required == TRUE)
    {
        current->rollback();
    }
    else // commit required
    {
        current->commit(/* report_heuristics = */ TRUE);
    }
}
catch (CORBA::TRANSACTION_ROLLEDBACK &exc)
{
    // The application called commit, but the transaction service
rolled
    // the transaction back because an error was detected.
    ...
}
catch (CosTransactions::HeuristicMixed &exc)
{
    // The transaction service has reported that some or all of
the
    // resourceobjects have made a heuristic decision. This has
    // resulted in heuristic damage.
    ...
}
catch (CosTransactions::HeuristicHazard &exc)
{
    // The transaction service has reported that not all of the
determining the
    // resource objects could participate properly in
    // outcome of the transaction. There is a possibility of
    // heuristic damage.
    ...
}
catch (CORBA::UserException &exc)
{
    // Another type of user exception has occurred.
    ...
}
catch (CORBA::SystemException &exc)
{
    // The application called commit, but the transaction
service
    // rolledthe transaction back because an error was detected.
    ...
}
catch (...)
{
    // A general exception has occurred.
    ...
}
...

```

Java Example

```

import org.omg.CosTransactions.*; // CosTransactions module
...
org.omg.CORBA.Boolean rollback_required = FALSE;
...
//Access the org.omg.CosTransactions.Current object.
org.omg.CORBA.Object orbCurrentPtr =
    com.ibm.CBCUtil.CBSeriesGlobal.orb().resolve_initial_references(
        "TransactionCurrent");
org.omg.CosTransactions.Current current =
    org.omg.CosTransactions.CurrentHelper.narrow(orbCurrentPtr);
// Invoke the begin operation on the org.omg.CosTransactions.Current
object.

```

```

current.begin();
// Perform work for the transaction and set rollback_required to
TRUE if
// an error is detected.
...
// Invoke commit or rollback depending on whether rollback_required
is
// set. This must be called within a try...catch structure as the
// transaction service may raise an exception if an error occurs.
try
{
    if (rollback_required == TRUE)
    {
        current.rollback();
    }
    else // commit required
    {
        current.commit(/* report_heuristics = */ TRUE);
    }
}
catch (org.omg.CORBA.TRANSACTION_ROLLEDBACK exc)
{
    // The application called commit, but the transaction service
    rolled
    // the transaction back because an error was detected.
    ...
}
catch (org.omg.CosTransactions.HeuristicMixed exc)
{
    // The transaction service has reported that some or all of
    the
    // resourceobjects have made a heuristic decision. This has
    // resulted in heuristic damage.
    ...
}
catch (org.omg.CosTransactions.HeuristicHazard exc)
{
    // The transaction service has reported that not all of the
    determining the
    // resource objects could participate properly in
    // outcome of the transaction. There is a possibility of
    // heuristic damage.
    ...
}
catch (org.omg.CORBA.UserException exc)
{
    // Another type of user exception has occurred.
    ...
}
catch (org.omg.CORBA.SystemException exc)
{
    // The application called commit, but the transaction
    service
    // rolledthe transaction back because an error was detected.
    ...
}
catch (Exception exc)
{
    // A general exception has occurred.
    ...
}
...

```

Current::rollback_only

| | |
|---------------------------|---|
| Overview | Modifies the transaction such that it cannot be committed, but can only be rolled back. |
| Original interface | "CosTransactions::Current Interface" on page 263 |
| Exceptions | NoTransaction |

Intended Usage

If there is no transaction associated with the current thread, the NoTransaction exception is raised.

The effect of this request is equivalent to performing the ["rollback_only" on page 263](#) Operation in the corresponding ["Coordinator" on page 254](#) Interface.

IDL Syntax

```
void rollback_only() raises(NoTransaction);
```

Input parameters

None.

Return values

None.

Examples

The following examples demonstrate the usage of `CosTransactions::Current::rollback_only`.

C++ Example

```
#include <CosTransactions.hh> // CosTransactions module
...
//Access the CosTransactions::Current object.
CORBA::Object_ptr orbCurrentPtr =
    CBSeriesGlobal::orb()->resolve_initial_references("TransactionCurrent");
CosTransactions::Current_ptr current =
    CosTransactions::Current::_narrow(orbCurrentPtr);
// Invoke the rollback_only operation on the
CosTransactions::Current object.
current->rollback_only();
...
```

Java Example

```
import org.omg.CosTransactions.*; // CosTransactions module
...
//Access the org.omg.CosTransactions.Current object.
org.omg.CORBA.Object orbCurrentPtr =
    com.ibm.CBCUtil.CBSeriesGlobal.orb().resolve_initial_references(
        "TransactionCurrent");
org.omg.CosTransactions.Current current =
    org.omg.CosTransactions.CurrentHelper.narrow(orbCurrentPtr);
// Invoke the rollback_only operation on the
org.omg.CosTransactions.Current
// object.
current.rollback_only();
...
```

Current::set_timeout

| | |
|---------------------------|--|
| Overview | Sets the timeout value to be used for all subsequent transactions. |
| Original interface | "CosTransactions::Current Interface" on page |
| Exceptions | "INITIALIZE standard exception" on page |

Intended Usage

Subsequent transactions created are subject to being rolled back if they do not complete within the time limit specified on this parameter. The default value for the time limit is platform dependent. If the parameter is zero, there is no application-specific time limit.

The INITIALIZE standard exception is raised if `set_timeout` is being used for the first time and the Transaction Service cannot be initialized.

IDL Syntax

```
void set_timeout(in unsigned long seconds);
```

Input parameters

seconds

The value of the time limit in seconds.

Return values

None.

Examples

The following examples demonstrate the usage of `CosTransactions::Current::set_timeout`.

C++ Example

```
#include <CosTransactions.hh> // CosTransactions module
...
//Access the CosTransactions::Current object.
CORBA::Object_ptr orbCurrentPtr =
    CBSeriesGlobal::orb()->resolve_initial_references("TransactionCurrent");
CosTransactions::Current_ptr current =
    CosTransactions::Current::_narrow(orbCurrentPtr);
// Invoke the set_timeout operation on the CosTransactions::Current
object.
current->set_timeout(60 /* seconds */);
...
// Start a transaction
...
```

Java Example

```
import org.omg.CosTransactions.*; // CosTransactions module
...
//Access the org.omg.CosTransactions.Current object.
org.omg.CORBA.Object orbCurrentPtr =
    com.ibm.CBCUtil.CBSeriesGlobal.orb().resolve_initial_references(
        "TransactionCurrent");
org.omg.CosTransactions.Current current =
    org.omg.CosTransactions.CurrentHelper.narrow(orbCurrentPtr);
// Invoke the set_timeout operation on the
org.omg.CosTransactions.Current
// object.
current.set_timeout(60 /* seconds */);
...
// Start a transaction
...
```

Current::suspend

| | |
|---------------------------|---|
| Overview | Returns an object that represents the transaction context currently associated with the current thread, and disassociates the currently associated transaction context from the current thread. |
| Original interface | "CosTransactions::Current Interface" on page |

Intended Usage

If there is no current transaction, a NULL reference is returned.

This object can be given to the ["resume" on page 270](#) operation to reestablish this context in the same, or a different, thread within the same server process.

IDL Syntax

```
Control suspend();
```

Input parameters

Control

Represents the transaction context currently associated with the current thread. The caller should not free the returned object; the Transaction Service retains ownership of it.

Return values

None.

Examples

The following examples demonstrate the usage of `CosTransactions::Current::suspend`.

C++ Example

```

#include <CosTransactions.hh> // CosTransactions module
...
CosTransactions::Control_ptr control = NULL;
//Access the CosTransactions::Current object.
CORBA::Object_ptr orbCurrentPtr =
    CBSeriesGlobal::orb()->resolve_initial_references("TransactionCurrent");
CosTransactions::Current_ptr current =
    CosTransactions::Current::_narrow(orbCurrentPtr);
...
// Invoke the begin operation on the CosTransactions::Current
object.
current->begin();
...
// Suspend the association between the transaction and the thread.
control = current->suspend();
if (!control)
{
    // There was no transaction associated with this thread prior to
the
    // suspend. Perform appropriate action.
    cout << "Error: No transaction prior to suspend" << endl;
}
}

```

Java Example

```

import org.omg.CosTransactions.*; // CosTransactions module
...
org.omg.CosTransactions.Control control = null;
//Access the org.omg.CosTransactions.Current object.
org.omg.CORBA.Object orbCurrentPtr =
    com.ibm.CBCUtil.CBSeriesGlobal.ORB().resolve_initial_references(
        "TransactionCurrent");
org.omg.CosTransactions.Current current =
    org.omg.CosTransactions.CurrentHelper.narrow(orbCurrentPtr);
...
// Invoke the begin operation on the org.omg.CosTransactions.Current
object.
current.begin();
...
// Suspend the association between the transaction and the thread.
control = current.suspend();
if (control == null)
{
    // There was no transaction associated with this thread prior to
the
    // suspend. Perform appropriate action.
    System.out.println ("Error: No transaction prior to suspend");
}
}

```

CosTransactions::RecoveryCoordinator Interface

This interface is not part of the programming model and should not be directly invoked or overridden.

CosTransactions::Resource Interface

This interface is not part of the programming model and should not be directly invoked or overridden.

CosTransactions::Synchronization Interface

This interface is not part of the programming model and should not be directly invoked or overridden.

Synchronization::after_completion

| | |
|-----------------|---|
| Overview | Informs the target object that the transaction that it represents has been completed. |
|-----------------|---|

| | |
|--------------------|--|
| Original interface | "CosTransactions::Synchronization Interface" on page 276 |
|--------------------|--|

Intended Usage

Called from within the Transaction Service. Informs the target object that the transaction that it represents has been completed. The current status of the transaction (as determined by the ["register_synchronization" on page 263](#) operation of the ["Coordinator" on page 254](#) Interface) is provided as input.

IDL Syntax

```
void after_completion(in Status status);
```

Input parameters

status

The current status of the transaction, as determined by the ["Coordinator::register_synchronization" on page 263](#) operation.

Return values

None.

Synchronization::before_completion

| | |
|--------------------|---|
| Overview | Informs the target object that the transaction that it represents is about to be completed. |
| Original interface | "CosTransactions::Synchronization Interface" on page 276 |

Intended Usage

Called from within the Transaction Service. Informs the target object that the transaction that it represents is about to be completed. The target object can perform transactional work before it returns.

IDL Syntax

```
void before_completion();
```

Input parameters

None.

Return values

None.

CosTransactions::Terminator Interface

| | |
|----------------------|--|
| Overview | Defines operations to complete a transaction, either by requesting commitment or demanding rollback. |
| File stem | CosTransactions |
| Exceptions | HeuristicHazard HeuristicMixed |
| Supported operations | "Terminator::commit" on page 278 "Terminator::rollback" on page 279 |

Intended Usage

Defines operations to complete a transaction, either by requesting commitment or

demanding rollback. Typically, these operations are used by the transaction originator. An object that supports the Terminator interface is implicitly associated with one transaction only.

IDL syntax

```
interface Terminator
{
    void commit(in boolean report_heuristics)
        raises(HeuristicMixed,
              HeuristicHazard);
    void rollback();
};
```

Terminator::commit

| | |
|--|---|
| Overview | Requests that the transaction be committed. |
| Original interface | "CosTransactions::Terminator Interface" on page 277 |
| Exceptions | HeuristicHazard |
| HeuristicMixed | |
| "TransactionRolledBack standard exception" on page | |

Intended Usage

If the transaction has not been marked as rollback-only, and all the participants in the transaction agree to commit, the transaction is committed, and the operation terminates normally. Otherwise, the transaction is rolled back and the standard TransactionRolledBack exception is raised.

If the report_heuristics parameter is true, the transaction framework reports inconsistent or possibly-inconsistent outcomes using the HeuristicMixed or HeuristicHazard exceptions.

The commit operation can rollback the transaction if there are existing or potential activities associated with the transaction that have not completed.

When a top-level transaction is committed, all changes to transactional objects made in the scope of this transaction are made permanent and visible to other transactions or clients.

Note that the ["suspend" on page 275](#) operation of the ["CosTransactions::Current Interface" on page](#) must be used to suspend an active transaction before the commit operation of the Terminator interface is used to commit the transaction.

IDL Syntax

```
void commit(in boolean report_heuristics)
    raises(HeuristicMixed, HeuristicHazard);
```

Input parameters

report_heuristics

Flag indicating whether heuristic reporting is required.

Return values

None.

Examples

The following examples demonstrate the usage of CosTransactions::Terminator::commit.

C++ Example

```
{
    try
```

```

CosTransactions::Control_var control;
CosTransactions::Terminator_var term;
.....
control = current->suspend();
term = control->get_terminator();
term->commit(TRUE);
}
catch ( .....
}

```

Java Example

```

{
try
{
org.omg.CosTransactions.Control control;
org.omg.CosTransactions.Terminator term;
.....
control = current.suspend();
term = control.get_terminator();
term.commit(true);
}
catch ( .....
}

```

Terminator::rollback

| | |
|---------------------------|---|
| Overview | Demands that the transaction be rolled back. |
| Original interface | "CosTransactions::Terminator Interface" on page 277 |

Intended Usage

When a transaction is rolled back, all changes to transactional objects made in the scope of this transaction (including changes made by descendant transactions) are rolled back. All resources locked by the transaction are made available to other transactions as appropriate to the degree of isolation enforced by the resources.

Note that the ["suspend" on page 275](#) operation of the ["CosTransactions::Current Interface" on page](#) must be used to suspend an active transaction before the rollback operation of the Terminator interface is used to rollback the transaction.

IDL Syntax

```
void rollback();
```

Input parameters

None.

Return values

None.

Examples

The following examples demonstrate the usage of CosTransactions::Terminator::rollback.

C++ Example

```

{
try
{
CosTransactions::Control_var control;
CosTransactions::Terminator_var term;
.....
control = current->suspend();
term = control->get_terminator();
term->rollback();
}
catch ( .....
}

```

Java Example

```

{
try

```

```

    {
      org.omg.CosTransactions.Control control;
      org.omg.CosTransactions.Terminator term;
      .....
      control = current.suspend();
      term = control.get_terminator();
      term.rollback();
    }
  } catch ( .....
}

```

CosTransactions::TransactionalObject Interface

| | |
|------------------|---|
| Overview | Used by an object to indicate that it is transactional. |
| File stem | CosTransactions |

Intended Usage

The TransactionalObject interface is used by an object to indicate that it is transactional. By inheriting from the TransactionalObject interface, an object indicates that it wants the transaction context associated with the client thread to be propagated on requests to the object.

The TransactionalObject interface defines no operations. It is simply a marker.

IDL syntax

```
interface TransactionalObject{}
```

CosTransactions::TransactionFactory Interface

This interface is not part of the programming model and should not be directly invoked or overridden.

C++ value type library, methods implemented

The WebSphere valuetype library for C++ implements the methods listed below:

- Javax::rmi::CORBA::ClassDesc
- Java::lang::Throwable

```
::CORBA::WStringValue* localizedMessage ()  
::CORBA::WStringValue* message ()  
::CORBA::Void setMessage (const ::CORBA::WStringValue& arg0);  
::CORBA::WStringValue* getMessage () const;  
::CORBA::WStringValue* toString ()
```

- java::lang::Exception
- java::io::IOException
- java::io::OutputStream

```
virtual ::CORBA::Void close ()  
virtual ::CORBA::Void flush ()  
//A pure virtual method  
virtual ::CORBA::Void write__long (::CORBA::Long arg0)= 0  
::CORBA::Void write__org_omg_boxedRMI_seq1_octet (::org::omg::boxedRMI::seq1_octet*  
arg0)  
::CORBA::Void write__org_omg_boxedRMI_seq1_octet__long__long  
(::org::omg::boxedRMI::seq1_octet* arg0, ::CORBA::Long  
arg2)
```

- java::io::FilterOutputStream

```
::CORBA::Void close ()  
::CORBA::Void flush ()  
::CORBA::Void write__long (::CORBA::Long arg0)  
::CORBA::Void write__org_omg_boxedRMI_seq1_octet (::org::omg::boxedRMI::seq1_octet*  
arg0)  
::CORBA::Void write__org_omg_boxedRMI_seq1_octet__long__long  
(::org::omg::boxedRMI::seq1_octet* arg0, ::CORBA::Long  
arg2)
```

- java::io::PrintStream

```
CORBA::Boolean checkError ()  
::CORBA::Void print__wchar (::CORBA::WChar arg0)  
::CORBA::Void print__double (::CORBA::Double arg0)  
::CORBA::Void print__float (::CORBA::Float arg0)  
::CORBA::Void print__long (::CORBA::Long arg0)  
::CORBA::Void print__long__long (::CORBA::LongLong arg0)  
::CORBA::Void print__java_lang_Object (const ::java::lang::Object& arg0)  
::CORBA::Void print__CORBA_WStringValue (::CORBA::WStringValue* arg0)  
::CORBA::Void print__boolean (::CORBA::Boolean arg0)  
::CORBA::Void print__org_omg_boxedRMI_seq1_wchar (::org::omg::boxedRMI::seq1_wchar*  
arg0)  
::CORBA::Void println__ ()  
::CORBA::Void println__wchar (::CORBA::WChar arg0)  
::CORBA::Void println__double (::CORBA::Double arg0)  
::CORBA::Void println__float (::CORBA::Float arg0)  
::CORBA::Void println__long (::CORBA::Long arg0)  
::CORBA::Void println__long__long (::CORBA::LongLong arg0)  
::CORBA::Void println__java_lang_Object (const ::java::lang::Object& arg0)  
::CORBA::Void println__CORBA_WStringValue (::CORBA::WStringValue* arg0)  
::CORBA::Void println__boolean (::CORBA::Boolean arg0)  
::CORBA::Void println__org_omg_boxedRMI_seq1_wchar  
(::org::omg::boxedRMI::seq1_wchar* arg0)  
::CORBA::Void setError ()
```

- java::io::Writer

```
virtual ::CORBA::Void close ()  
virtual ::CORBA::Void flush ()  
::CORBA::Void write__long(CORBA::Long arg0)  
::CORBA::Void write__CORBA_WStringValue (::CORBA::WStringValue* arg0)  
::CORBA::Void write__CORBA_WStringValue__long__long (::CORBA::WStringValue* arg0,  
::CORBA::Long arg1, ::CORBA::Long arg2)  
::CORBA::Void write__org_omg_boxedRMI_seq1_wchar (::org::omg::boxedRMI::seq1_wchar*  
arg0)  
::CORBA::Void write__org_omg_boxedRMI_seq1_wchar__long__long (  
::org::omg::boxedRMI::seq1_wchar* arg0,  
::CORBA::Long arg1, ::CORBA::Long arg2)
```

- java::io::PrintWriter

```
::CORBA::Boolean checkError ()  
::CORBA::Void print__wchar (::CORBA::WChar arg0)  
::CORBA::Void print__double (::CORBA::Double arg0)  
::CORBA::Void print__float (::CORBA::Float arg0)  
::CORBA::Void print__long (::CORBA::Long arg0)  
::CORBA::Void print__long__long (::CORBA::LongLong arg0)  
::CORBA::Void print__java_lang_Object (const ::java::lang::Object& arg0)  
::CORBA::Void print__CORBA_WStringValue (::CORBA::WStringValue* arg0)
```

```

::CORBA::Void print_boolean (::CORBA::Boolean arg0)
::CORBA::Void print_org_omg_boxedRMI_seq1_wchar (::org::omg::boxedRMI::seq1_wchar*
arg0)
::CORBA::Void println_()
::CORBA::Void println_wchar (::CORBA::WChar arg0)
::CORBA::Void println_double (::CORBA::Double arg0)
::CORBA::Void println_float (::CORBA::Float arg0)
::CORBA::Void println_long (::CORBA::Long arg0)
::CORBA::Void println_long_long (::CORBA::LongLong arg0)
::CORBA::Void println_java_lang_Object (const ::java::lang::Object& arg0)
::CORBA::Void println_CORBA_WStringValue (::CORBA::WStringValue* arg0)
::CORBA::Void println_boolean (::CORBA::Boolean arg0)
::CORBA::Void println_org_omg_boxedRMI_seq1_wchar
(::org::omg::boxedRMI::seq1_wchar* arg0)
::CORBA::Void setError ()

```

- javax::ejb::CreateException
- javax::ejb::RemoveException
- javax::ejb::FinderException
- javax::ejb::ObjectNotFoundException
- javax::ejb::DuplicateKeyException
- javax::ejb::EJBMetaData

```

::javax::ejb::EJBHome_ptr getEJBHome ()
::javax::rmi::CORBA::ClassDesc* getHomeInterfaceClass ()
::javax::rmi::CORBA::ClassDesc* getRemoteInterfaceClass ()
::javax::rmi::CORBA::ClassDesc* getPrimaryKeyClass ()
void setEJBHome (::javax::ejb::EJBHome_ptr arg0);
void setHomeInterfaceClass (::javax::rmi::CORBA::ClassDesc* arg0);
void setRemoteInterfaceClass (::javax::rmi::CORBA::ClassDesc* arg0);
void setPrimaryKeyClass (::javax::rmi::CORBA::ClassDesc* arg0)
::CORBA::Boolean isSession ()

```

- java::util::Vector

```

::CORBA::Long getCapacity() const;
::CORBA::Void setCapacity (::CORBA::Long cap);
java_util_Vector_Impl& operator=(const java_util_Vector_Impl& aVector);
const std::vector<java::lang::Object>& getVectorInstance() const;
const std::vector<java::lang::Object>::iterator& getObjectIterator() const;
std::vector<java::lang::Object>::iterator& resetObjectIterator();
::CORBA::Void setCapacityIncrement (::CORBA::Long incrementValue);
::CORBA::Long getCapacityIncrement();
::CORBA::Void addElement (const ::java::lang::Object& arg0);
::CORBA::Long capacity ();
::java::lang::Object* clone ();
java::util::Vector* cloneVector ();
::CORBA::Void copyInto (::org::omg::boxedRMI::java::lang::seq1_Object* arg0);
::java::lang::Object* elementAt (::CORBA::Long arg0);
::java::lang::Object* getElements ();
::CORBA::Void ensureCapacity (::CORBA::Long arg0);
::java::lang::Object* firstElement ();
::CORBA::Long indexOf__java_lang_Object__long (const ::java::lang::Object& arg0,
::CORBA::Long arg1);
::CORBA::Void insertElementAt (const ::java::lang::Object& arg0, ::CORBA::Long
arg1);
::CORBA::Boolean isEmpty();
::java::lang::Object* lastElement ();
::CORBA::Long lastIndexOf__java_lang_Object__long (const ::java::lang::Object& arg0,
::CORBA::Long arg1);
::CORBA::Void removeAllElements ();
::CORBA::Boolean removeElement (const ::java::lang::Object& arg0);
::CORBA::Void removeElementAt (::CORBA::Long arg0);
::CORBA::Void setElementAt (const ::java::lang::Object& arg0, ::CORBA::Long arg1);
::CORBA::Void setSize (::CORBA::Long arg0);
::CORBA::Long size();
::CORBA::Void trimToSize ();

```

- java::lang::Number

```

virtual ::CORBA::Long intValue();
virtual ::CORBA::LongLong longValue();
virtual ::CORBA::Float floatValue();
virtual ::CORBA::Double doubleValue();
virtual ::CORBA::Octet byteValue();
virtual ::CORBA::Short shortValue();

```

- java::lang::Boolean

```

::CORBA::Boolean booleanValue ();
::CORBA::Boolean equals (const ::java::lang::Object& arg0);
::CORBA::Boolean getBoolean (::CORBA::WStringValue* arg0);
::CORBA::Long hashCode ();
::CORBA::WStringValue* toString ();
::java::lang::Boolean* valueOf (::CORBA::WStringValue* arg0);

```

- java::lang::Byte

```

::CORBA::Octet      byteValue ();
::CORBA::Long      compareTo (const ::java::lang::Object& arg0);
::CORBA::Long      compareTo__java_lang_Byte (::java::lang::Byte* arg0);
::java::lang::Byte* decode (::CORBA::WStringValue* arg0);
::CORBA::Boolean   equals (const ::java::lang::Object& arg0);
::CORBA::Long      hashCode ();
::CORBA::Octet     parseByte__CORBA_WStringValue (::CORBA::WStringValue* arg0);
::CORBA::Octet     parseByte__CORBA_WStringValue__long (::CORBA::WStringValue*
arg0, ::CORBA::Long
arg1);
::CORBA::WStringValue* toString__ ();
::CORBA::WStringValue* toString__octet (::CORBA::Octet arg0);
::java::lang::Byte* valueOf__CORBA_WStringValue (::CORBA::WStringValue* arg0);
::java::lang::Byte* valueOf__CORBA_WStringValue__long (::CORBA::WStringValue*
arg0,
::CORBA::Long arg1);

```

- **java::lang::Character**

```

::CORBA::WChar      charValue ();
::CORBA::Long      compareTo (const ::java::lang::Object& arg0);
::CORBA::Long      compareTo__java_lang_Character (::java::lang::Character*
arg0);
::CORBA::Long      digit (::CORBA::WChar arg0, ::CORBA::Long arg1);
::CORBA::Boolean   equals (const ::java::lang::Object& arg0);
::CORBA::WChar     forDigit (::CORBA::Long arg0, ::CORBA::Long arg1);
::CORBA::Long      getNumericValue (::CORBA::WChar arg0);
::CORBA::Long      getType (::CORBA::WChar arg0);
::CORBA::Long      hashCode ();
::CORBA::Boolean   isDefined (::CORBA::WChar arg0);
::CORBA::Boolean   isDigit (::CORBA::WChar arg0);
::CORBA::Boolean   isISOControl (::CORBA::WChar arg0);
::CORBA::Boolean   isIdentifierIgnorable (::CORBA::WChar arg0);
::CORBA::Boolean   isJavaIdentifierPart (::CORBA::WChar arg0);
::CORBA::Boolean   isJavaIdentifierStart (::CORBA::WChar arg0);
::CORBA::Boolean   isJavaLetter (::CORBA::WChar arg0);
::CORBA::Boolean   isJavaLetterOrDigit (::CORBA::WChar arg0);
::CORBA::Boolean   isLetter (::CORBA::WChar arg0);
::CORBA::Boolean   isLetterOrDigit (::CORBA::WChar arg0);
::CORBA::Boolean   isLowerCase (::CORBA::WChar arg0);
::CORBA::Boolean   isSpace (::CORBA::WChar arg0);
::CORBA::Boolean   isSpaceChar (::CORBA::WChar arg0);
::CORBA::Boolean   isTitleCase (::CORBA::WChar arg0);
::CORBA::Boolean   isUnicodeIdentifierPart (::CORBA::WChar arg0);
::CORBA::Boolean   isUnicodeIdentifierStart (::CORBA::WChar arg0);
::CORBA::Boolean   isUpperCase (::CORBA::WChar arg0);
::CORBA::Boolean   isWhitespace (::CORBA::WChar arg0);
::CORBA::WChar     toLowerCase (::CORBA::WChar arg0);
::CORBA::WStringValue* toString ();
::CORBA::WChar     toTitleCase (::CORBA::WChar arg0);
::CORBA::WChar     toUpperCase (::CORBA::WChar arg0);

```

- **java::lang::Double**

```

::CORBA::Long      compareTo (const ::java::lang::Object& arg0);
::CORBA::Long      compareTo__java_lang_Double (::java::lang::Double* arg0);
::CORBA::LongLong doubleToLongBits (::CORBA::Double arg0);
::CORBA::Double    doubleValue ();
::CORBA::Boolean   equals (const ::java::lang::Object& arg0);
::CORBA::Long      hashCode ();
::CORBA::Boolean   infinite ();
::CORBA::Boolean   isInfinite (::CORBA::Double arg0);
::CORBA::Boolean   NaN ();
::CORBA::Boolean   isNaN (::CORBA::Double arg0);
::CORBA::Double    longBitsToDouble (::CORBA::LongLong arg0);
::CORBA::Double    parseDouble (::CORBA::WStringValue* arg0);
::CORBA::WStringValue* toString__ ();
::CORBA::WStringValue* toString__double (::CORBA::Double arg0);
::java::lang::Double* valueOf (::CORBA::WStringValue* arg0);

```

- **java::lang::Float**

```

::CORBA::Long      compareTo (const ::java::lang::Object& arg0);
::CORBA::Long      compareTo__java_lang_Float (::java::lang::Float* arg0);
::CORBA::Boolean   equals (const ::java::lang::Object& arg0);
::CORBA::Long      floatToIntBits (::CORBA::Float arg0);
::CORBA::Float     floatValue ();
::CORBA::Long      hashCode ();
::CORBA::Float     intBitsToFloat (::CORBA::Long arg0);
::CORBA::Boolean   infinite ();
::CORBA::Boolean   isInfinite (::CORBA::Float arg0);
::CORBA::Boolean   NaN ();
::CORBA::Boolean   isNaN (::CORBA::Float arg0);
::CORBA::Float     parseFloat (::CORBA::WStringValue* arg0);
::CORBA::WStringValue* toString__ ();
::CORBA::WStringValue* toString__float (::CORBA::Float arg0);
::java::lang::Float* valueOf (::CORBA::WStringValue* arg0);

```

- **java::lang::Integer**

```

::CORBA::Long      compareTo (const ::java::lang::Object& arg0);
::CORBA::Long      compareTo__java_lang_Integer (::java::lang::Integer* arg0);
::java::lang::Integer* decode (::CORBA::WStringValue* arg0);
::CORBA::Boolean   equals (const ::java::lang::Object& arg0);
::java::lang::Integer* getInteger__CORBA_WStringValue (::CORBA::WStringValue*
arg0);
::java::lang::Integer* getInteger__CORBA_WStringValue__long
 (::CORBA::WStringValue*
arg0, ::CORBA::Long arg1);

```

```

::java::lang::Integer*   getInteger__CORBA_WStringValue__java_lang_Integer
                        (::CORBA::WStringValue* arg0, ::java::lang::Integer*
arg1);
::CORBA::Long           hashCode ();
::CORBA::Long           intValue ();
::CORBA::Long           parseInt__CORBA_WStringValue (::CORBA::WStringValue* arg0);
::CORBA::Long           parseInt__CORBA_WStringValue__long (::CORBA::WStringValue*
arg0,
                        ::CORBA::Long
arg1);
::CORBA::WStringValue* toBinaryString (::CORBA::Long arg0);
::CORBA::WStringValue* toHexString (::CORBA::Long arg0);
::CORBA::WStringValue* toOctalString (::CORBA::Long arg0);
::CORBA::WStringValue* toString__ ();
::CORBA::WStringValue* toString__long (::CORBA::Long arg0);
::CORBA::WStringValue* toString__long__long (::CORBA::Long arg0, ::CORBA::Long
arg1);
::java::lang::Integer*  valueOf__CORBA_WStringValue (::CORBA::WStringValue* arg0);
::java::lang::Integer*  valueOf__CORBA_WStringValue__long (::CORBA::WStringValue*
arg0,
                        ::CORBA::Long
arg1);

```

- **java::lang::Long**

```

::CORBA::Long           ompareTo (const ::java::lang::Object& arg0);
::CORBA::Long           compareTo__java_lang_Long (::java::lang::Long* arg0);
::java::lang::Long*    decode (::CORBA::WStringValue* arg0);
::CORBA::Boolean       equals (const ::java::lang::Object& arg0);
::java::lang::Long*    getLong__CORBA_WStringValue (::CORBA::WStringValue* arg0);
::java::lang::Long*    getLong__CORBA_WStringValue__long__long
(::CORBA::WStringValue*
                        arg0, ::CORBA::LongLong arg1);
::java::lang::Long*    getLong__CORBA_WStringValue__java_lang_Long
                        (::CORBA::WStringValue* arg0, ::java::lang::Long*
arg1);
::CORBA::Long           hashCode ();
::CORBA::LongLong      longValue ();
::CORBA::LongLong      parseLong__CORBA_WStringValue (::CORBA::WStringValue* arg0);
::CORBA::LongLong      parseLong__CORBA_WStringValue__long (::CORBA::WStringValue*
arg0, ::CORBA::Long arg1);
::CORBA::WStringValue* toBinaryString (::CORBA::LongLong arg0);
::CORBA::WStringValue* toHexString (::CORBA::LongLong arg0);
::CORBA::WStringValue* toOctalString (::CORBA::LongLong arg0);
::CORBA::WStringValue* toString__ ();
::CORBA::WStringValue* toString__long__long (::CORBA::LongLong arg0);
::CORBA::WStringValue* toString__long__long__long (::CORBA::LongLong arg0,
                        ::CORBA::Long
arg1);
::java::lang::Long*    valueOf__CORBA_WStringValue (::CORBA::WStringValue* arg0);
::java::lang::Long*    valueOf__CORBA_WStringValue__long (::CORBA::WStringValue*
arg0,
                        ::CORBA::Long
arg1);

```

- **java::lang::Short**

```

::CORBA::Long           compareTo (const ::java::lang::Object& arg0);
::CORBA::Long           compareTo__java_lang_Short (::java::lang::Short* arg0);
::java::lang::Short*   decode (::CORBA::WStringValue* arg0);
::CORBA::Boolean       equals (const ::java::lang::Object& arg0);
::CORBA::Long           hashCode ();
::CORBA::Short         parseShort__CORBA_WStringValue (::CORBA::WStringValue* arg0);
::CORBA::Short         parseShort__CORBA_WStringValue__long (::CORBA::WStringValue*
arg0, ::CORBA::Long arg1);
::CORBA::Short         shortValue ();
::CORBA::WStringValue* toString__ ();
::CORBA::WStringValue* toString__short (::CORBA::Short arg0);
::java::lang::Short*   valueOf__CORBA_WStringValue (::CORBA::WStringValue* arg0);
::java::lang::Short*   valueOf__CORBA_WStringValue__long (::CORBA::WStringValue*
arg0,
                        ::CORBA::Long
arg1);

```

- **java::lang::Throwable_init**

```

virtual CORBA::ValueBase *create_for_unmarshal()
virtual java::lang::Throwable* create__ ()
virtual java::lang::Throwable* create__CORBA_WStringValue (::CORBA::WStringValue*
arg0)

```

- **java::lang::Exception_init**

```

virtual CORBA::ValueBase *create_for_unmarshal()
virtual java::lang::Exception* create__ ()
virtual java::lang::Exception* create__CORBA_WStringValue (::CORBA::WStringValue*
arg0)

```

- **java::io::IOException_init**

```

virtual CORBA::ValueBase *create_for_unmarshal()
virtual java::io::IOException* create__ ()
virtual java::io::IOException* create__CORBA_WStringValue (::CORBA::WStringValue*
arg0)

```

- javax::ejb::CreateException_init

```
virtual CORBA::ValueBase *create_for_unmarshal()
virtual javax::ejb::CreateException* create__ ()
virtual javax::ejb::CreateException* create__CORBA_WStringValue
 (::CORBA::WStringValue* arg0)
```

- javax::ejb::RemoveException_init

```
virtual CORBA::ValueBase *create_for_unmarshal()
virtual javax::ejb::RemoveException* create__ ()
virtual javax::ejb::RemoveException* create__CORBA_WStringValue
 (::CORBA::WStringValue*arg0)
```

- javax::ejb::FinderException_init

```
virtual CORBA::ValueBase *create_for_unmarshal()
virtual javax::ejb::RemoveException* create__ ()
virtual javax::ejb::RemoveException* create__CORBA_WStringValue
 (::CORBA::WStringValue*arg0)
```

- javax::ejb::ObjectNotFoundException_init

```
virtual CORBA::ValueBase *create_for_unmarshal()
virtual javax::ejb::RemoveException* create__ ()
virtual javax::ejb::RemoveException*
create__CORBA_WStringValue (::CORBA::WStringValue*arg0)
```

- javax::ejb::DuplicateKeyException_init

```
virtual CORBA::ValueBase *create_for_unmarshal()
virtual javax::ejb::RemoveException* create__ () virtual
javax::ejb::RemoveException* create__CORBA_WStringValue (::CORBA::WStringValue*arg0)
```

- java::util::Vector_init

```
virtual ::CORBA::ValueBase *create_for_unmarshal()
virtual ::java::util::Vector* create__ ()
::java::util::Vector* create__long (::CORBA::Long arg0)
::java::util::Vector* create__long_long (::CORBA::Long arg0, ::CORBA::Long arg1)
```

- com::ibm::ws::java_io_PrintStream_factory

```
virtual CORBA::ValueBase *create_for_unmarshal()
virtual ::java::io::PrintStream* create__ ()
virtual ::java::io::PrintStream* create__java_io_OutputStream (
::java::io::OutputStream *arg0)
virtual ::java::io::PrintWriter* create__java_io_Writer (::java::io::Writer *arg0)
/**
 * Create a new print stream over a file output stream.
 *
 * @param The name of the file output stream to which values and objects will be
 * printed.
 * @return the pointer to the created PrintStream object.
 */
virtual ::java::io::PrintStream* create__CORBA_WStringValue (::CORBA::WStringValue*
arg0)
```

- com::ibm::ws::java_io_FilterOutputStream_factory

```
virtual CORBA::ValueBase *create_for_unmarshal()
virtual ::java::io::FilterOutputStream* create__ ()
```

- com::ibm::ws::java_io_PrintWriter_factory

```
virtual CORBA::ValueBase *create_for_unmarshal()
virtual ::java::io::PrintWriter* create__ ()
virtual ::java::io::PrintWriter* create__java_io_Writer (::java::io::Writer *arg0)
virtual ::java::io::PrintWriter* create__java_io_OutputStream (
::java::io::OutputStream *arg0)
/**
 * Create a new print writer over a file output stream.
 *
 * @param The name of the file output stream to which values and objects will be
 * printed.
 * @return the pointer to the created PrintStream object.
 */
virtual ::java::io::PrintWriter* create__CORBA_WStringValue (::CORBA::WStringValue*
arg0)
```

- com::ibm::ws::javax_rmi_CORBA_ClassDesc_factory

```
virtual CORBA::ValueBase *create_for_unmarshal()
virtual javax::rmi::CORBA::ClassDesc *create__ ()
```

- com::ibm::ws::javax_ejb_EJBMetaData_factory

```
virtual CORBA::ValueBase *create_for_unmarshal()
virtual ::javax::ejb::EJBMetaData *create__()
```

- **java::lang::Boolean_init**

```
virtual CORBA::ValueBase *create_for_unmarshal();
java::lang::Boolean* create__CORBA_WStringValue(::CORBA::WStringValue* arg0);
java::lang::Boolean* create__boolean(CORBA::Boolean arg0);
```

- **java::lang::Byte_init**

```
virtual CORBA::ValueBase *create_for_unmarshal();
java::lang::Byte* create__();
java::lang::Byte* create__octet(::CORBA::Octet arg0);
java::lang::Byte* create__CORBA_WStringValue(::CORBA::WStringValue* arg0);
```

- **java::lang::Integer_init**

```
virtual CORBA::ValueBase *create_for_unmarshal();
java::lang::Integer* create__long (::CORBA::Long arg0);
java::lang::Integer* create__CORBA_WStringValue (::CORBA::WStringValue* arg0);
```

- **java::lang::Short_init**

```
virtual CORBA::ValueBase *create_for_unmarshal();
java::lang::Short* create__CORBA_WStringValue (::CORBA::WStringValue* arg0);
java::lang::Short* create__short (::CORBA::Short arg0);
```

- **java::lang::Long_init**

```
virtual CORBA::ValueBase *create_for_unmarshal();
java::lang::Long* create__long_long (::CORBA::LongLong arg0);
java::lang::Long* create__CORBA_WStringValue (::CORBA::WStringValue* arg0);
```

- **java::lang::Float_init**

```
virtual CORBA::ValueBase *java_lang_Float_factory::create_for_unmarshal();
java::lang::Float *java_lang_Float_factory::create__double (::CORBA::Double arg0);
java::lang::Float *java_lang_Float_factory::create__float (::CORBA::Float arg0);
java::lang::Float *java_lang_Float_factory::create__CORBA_WStringValue
 (::CORBA::WStringValue* arg0);
```

- **java::lang::Double_init**

```
virtual CORBA::ValueBase *java_lang_Double_factory::create_for_unmarshal();
java::lang::Double *java_lang_Double_factory::create__double (::CORBA::Double arg0);
java::lang::Double *java_lang_Double_factory::create__CORBA_WStringValue
 (::CORBA::WStringValue* arg0);
```

- **java::lang::Character_init**

```
virtual CORBA::ValueBase *create_for_unmarshal();
virtual java::lang::Character* create (::CORBA::WChar arg0);
```

- **com::ibm::ws::VtlUtil**

```
static const char* exceptionLogFileName;
static const char* debugLogFileName;
/**
 * debugOn set to 1 is the debugging mode.
 * debugOn set to 0 is the non-debugging mode.
 */
static const int debugOn; // = 0;
/**
 * debugInfoToStdOut set to 1, the debugging messages will be printed to stdout.
 * debugInfoToStdOut set to 0, the debugging messages will not be printed to stdout.
 */
static const ::CORBA::Boolean debugInfoToStdOut;
/**
 * debugInfoToFile set to 1, the debugging messages will be printed to the file
 defined by
 * debugLogFileName.
 * debugInfoToFile set to 0, the debugging messages will not be printed to a file.
 */
static const ::CORBA::Boolean debugInfoToFile; // = false;
/**
 * Print the debugging message string msg to the designated media when debugOn is
 true.
 *
 * @param msg the <code>char *</code> to be printed.
 * @return void
 */
static void debug(char *msg);
/**
 * Concatenate strings msg1 and msg2. Print the result string to the designated media
 if debugOn is true.
 */
```

```

* @param msg1 the <code>char *</code> to be printed.
* @param msg2 the <code>char *</code> to be printed.
* @return void
*/
static void debug(char *msg1, char *msg2);
/**
* Print the debugging message string str and the attributes of the exception e to
the designated media
* if debugOn is true.
*
* @param msg the <code>char *</code> to be printed.
* @param e the <code>java::lang::Throwable* </code> to be printed.
* @return void
*/
static void debug(char msg, java::lang::Throwable* e);
/**
* Print the attributes of the exception e to stderr and the designated log file
defined by the
* exceptionLogFileName.
*
* @param e the <code> java::lang::Throwable*</code> to be printed.
* @return void
*/
static void handleException(java::lang::Throwable* e);
/**
* Print the attributes of the exception e and the message string msg to stderr and
the designated log file * defined by the exceptionLogFileName.
*
* @param e the <code> java::lang::Throwable*</code> to be printed.
* @param msg the <code>char *</code> to be printed.
* @return void
*/
static void handleException(java::lang::Throwable* e, char *msg);
* Transform the string str to a WStringValue object and return its pointer.
*
* @param str the <code>char *</code> to be transformed.
* @return pointer to the transformed WStringValue object
*/
static ::CORBA::WStringValue* toWStringValue(const char *str);
/**
* Concatenate strings str1 and str2, and transform the result string to a
WStringValue object and return
* its pointer.
*
* @param str1 the <code>char *</code> to be transformed.
* @param str2 the <code>char *</code> to be transformed.
* @return pointer to the transformed WStringValue object
*/
static ::CORBA::WStringValue* toWStringValue(const char *str1, const char
*str2);
/**
* Concatenate strings str1, str2, and str3, and transform the result string to a
WStringValue object and
* return its pointer.
*
* @param str1 the <code>char *</code> to be transformed.
* @param str2 the <code>char *</code> to be transformed.
* @param str3 the <code>char *</code> to be transformed.
* @return pointer to the transformed WStringValue object
*/
static ::CORBA::WStringValue* toWStringValue(const char *str1, const char *str2,
const char *str3);
/**
* Transform the WStringValue object wsv to a string and return the pointer to the
string.
*
* @param wsv the pointer to <code>::CORBA::WstringValue </code> to be transformed.
* @return pointer to the transformed string.
*/
static char* WStringValueToString(::CORBA::WStringValue *wsv);
/**
* Returns the registered factory object for the valuetype that has the designated
repository id.
* If the factory object is not found, a NULL pointer will be returned.
*
* @param the repository id of the factory to be retrieved as defined in the
Vtlib.idl file.
* @return the pointer to the registered factory.
*/
static ::CORBA::ValueFactoryBase* getFactory(const char * rid);
/**
* Each of the following methods returns the registered factory object for the
named valuetype.
* If the factory object is not found, a NULL pointer will be returned.
*
* @return the pointer to the registered factory.
*/
static java::lang::Boolean_init* getBooleanFactory();
static java::lang::Byte_init* getByteFactory();
static java::lang::Character_init* getCharacterFactory();
static java::lang::Double_init* getDoubleFactory();
static java::lang::Float_init* getFloatFactory();
static java::lang::Integer_init* getIntegerFactory();
static java::lang::Long_init* getLongFactory();
static java::lang::Short_init* getShortFactory();
static java::lang::Throwable_init* getThrowableFactory();
static java::lang::Exception_init* getExceptionFactory();
static java::io::IOException_init* getIOExceptionFactory();
static javax::ejb::CreateException_init * getCreateExceptionFactory();
static javax::ejb::RemoveException_init * getRemoveExceptionFactory();
static java::util::Vector_init* getVectorFactory();
static com::ibm::ws::javax_rmi_CORBA_ClassDesc_factory* getClassDescFactory();
static com::ibm::ws::java_io_PrintStream_factory*
getPrintStreamFactory();
static com::ibm::ws::java_io_FilterOutputStream_factory*

```

```
getFilterOutputStreamFactory();
    static com.ibm.ws.java_io_PrintWriter_factory*
getPrintWriterFactory();
    static com.ibm.ws.javax_ejb_EJBMetaData_factory*
getEJBMetaDataFactory();
```

Runtime properties for CORBA clients and servers

This topic provides reference information about the properties that you can set to control the runtime environment of C++ CORBA clients and servers. Each property is listed in the following form:

property_name=value_type

[*default_value*] A description of the property, where

property_name

is the name of the property

value_type

is the type of value that the property can have.

[*default value*]

is the default value of the property (only shown if the property has a default value).

Client and Server general ORB properties

You can specify the following general ORB properties for both clients and servers:

com.ibm.CORBA.nameServerHost=host_name

The name of the host on which the client's name server runs. This host name is used with the ***com.ibm.CORBA.nameServerPort*** property to access the name server.

com.ibm.CORBA.nameServerPort=port_number

[900] The number of the port that the name server uses to communicate with clients and servers. This property is an integer port number, in the range 0 through 65536. This port number is used with the ***com.ibm.CORBA.nameServerHost*** property to access the name server.

com.ibm.CORBA.protocolVersion=iiop_version

[1.2] The default GIOP/IOP protocol version that the ORB uses to export object references. This property enables WebSphere Application Server ORBs to interoperate with non-WebSphere ORBs that use the same format. This property can be set to 1.0, 1.1, or 1.2 (for IOP 1.0, IOP 1.1, or IOP 1.2 respectively).

Note: The C++ ORB may downgrade the level depending on responses from a remote server.

com.ibm.CORBA.requestTimeout=time_seconds

[30] The time, in seconds, that a request waits for a response before timing out and issuing an error that indicates NO RESPONSE. If this property is set to 0 (zero), requests wait indefinitely until a response is received.

This property is either 0 (zero, for an indefinite wait) or an integer number of seconds in the range 1 through 300 seconds.

For tracing and debugging, this property must be set to 0 (zero).

com.ibm.CORBA.eMNumThreads=number_threads

[3] The initial size of the thread pool that is created to push events to consumers. This property is an integer in the range 1 through 20.

com.ibm.CORBA.enableFilters=yes_or_no

[yes] Whether or not the client or server can use RAS request interceptors to help you map runtime problems back to a specific client. This property has one of the

following values:

Yes

Log entries (and trace entries if you have turned trace on) have additional 'UnitOfWork' information that can help you debug runtime problems. But that information comes with a performance penalty. When your application environment is fully debugged and deployed, you should consider turning off this option to disable RAS interceptor filters and thereby improve performance.

No

Log entries (and trace entries if you have turned trace on) do not have 'UnitOfWork' information.

By disabling the filters on a client, the server-side RAS trace and activity log entries do not contain the data needed to map an event back to that specific client. If you later decide that you need that information, enable the filters again in the client properties file then restart the client.

By turning off the filters on the server side, no RAS trace and activity log entries is mappable back to any specific client. If you later decide that you need that information, enable the filters again in the server then restart the server.

You can disable RAS request Interceptors to improve the performance of your WebSphere application environment, but should consider this only if you are confident that the environment has been stable for some time and that you need some extra performance. You can disable or enable RAS request interceptors in any combination of servers and clients. For maximum performance improvement, you should disable RAS request interceptors in both your clients and servers.

com.ibm.CORBA.maximumHops=*number_of_location_forwards*

[5] The maximum number of location forwards that the client or server should follow before aborting object location.

This property is an integer in the range 0 through 65536. A value of 0 (zero), indicates that the client or server should keep following location forwards indefinitely until the object is located.

Server-specific ORB properties

You can specify the following ORB properties for servers only:

com.ibm.CORBA.hostName=*host_name*

The hostname string that should be included in object references (IORs) exported by the server. The value is a TCP/IP hostname of up to 256 ASCII characters.

Normally, the C++ ORB uses the dotted-decimal form of the hostname, but this property can be used to override the dotted-decimal form with an alternate name. This might be useful in situations where the server is operating behind a firewall, and you do not want the dotted-decimal hostname published outside the firewall.

com.ibm.CORBA.serverListenPort=*port_number*

[0] The port number on which the server listens for incoming requests from clients. For example, this enables the server to support a static firewall scenario, in which the firewall enables use of a set of "secure" ports.

If you leave this property to default to 0 (zero), the server is automatically assigned a number for its listening port.

com.ibm.CORBA.threadPoolSize=*number_threads*

[5] The size of the ORB thread pool in which servant objects process method invocations. This property is an integer in the range 0 through 1000.

When the ORB receives a request, it activates a thread from the appropriate pool for the target object to service the request.

com.ibm.CORBA.threadStackSize=*number_bytes*

[65536] The size, in bytes, of the thread stack used when creating new threads. This property is an integer in the range 0 through 65536 bytes.

Client and server code page support properties

You can specify the following code page properties for clients and servers:

com.ibm.CORBA.translationEnabled=*yes_no*

[no] Whether or not the client or server should perform code set translation for character data received in remote messages. This property can have the following values:

no

Code set translation is not performed the com.ibm.CORBA.nativeWCharSet property is ignored.

yes

Code set translation is performed. Also, consider the following points:

- If you do not specify a value for the com.ibm.CORBA.nativeWCharSet property, then the ORB assumes that the user does not want to use wchar (wide character) data because a wchar code set was not specified.
- You should not use the default of LANG or LC_ALL, but should manually set an appropriate value.
- Code set translation is not supported by the IIOp 1.0 protocol.

com.ibm.CORBA.isoLatin1=*yes_no*

[no] Whether or not the ORB uses ISO-Latin1 as the default transmission code set for character data in remote requests.

This property is ignored if you set com.ibm.CORBA.translationEnabled=yes. You should only set this property if the ORB communicates with a remote process that uses the IIOp 1.0 protocol.

com.ibm.CORBA.nativeCharSet=*codesetnum*

[0] The number of the native OSF (Open Systems Foundation) code set used by applications for single-byte char and string data.

This property is either 0 (zero) or a decimal code set number. If set to 0, the number of the native OSF code set used is calculated by the ORB.

com.ibm.CORBA.nativeWCharSet=*codesetnum*

[0] The number of the native OSF (Open Systems Foundation) code set used by applications for wchar (wide character) and wstring (wide string) data. This property need be set only if both applications (on the server) and the ORB support wchar and wstring types. If it is set, then the C++ ORB supports wchar translations; if it is not set, then wchar data is copied without translation.

This property is either 0 (zero, if you do not want to use wchar data) or a decimal code set number.

Note: If the code set property com.ibm.CORBA.nativeWCharSet does not match the current system code set, a DATA_CONVERSION exception is logged in the ORB's activity log.

Client and server trace and activity log support properties

You can specify the following trace and activity log properties for clients and servers:

com.ibm.CORBA.activityLogMaxSize=

[100] The maximum size (in kilobytes) that the activity log can reach. If the activity log reaches this size, it will wrap around, with the oldest messages in the log being overwritten with any new messages. This property is an integer number of kilobytes in the range 0 through infinity. When setting this property make sure that there is enough room available for the activity log on the disk where this directory is located (the directory of the activity log is set by the **com.ibm.CORBA.activityLogDirectory** property).

com.ibm.CORBA.activityLogDirectory=

[] The name of the directory that is used to store the activity log for the related client or server. If this value is left blank then the default directory is used, *wasee_install\services*; where, *wasee_install* is the directory into which you installed WebSphere Application Server enterprise services on the host. When setting this property make sure that there is enough room available for the activity log on the disk where this directory is located (the size of the activity log is set by the **com.ibm.CORBA.activityLogMaxSize** property).

com.ibm.CORBA.orbCommunicationsTraceLevel=trace_level

[none] Controls the amount of trace data that is written to the trace log for communications between the Object Request Broker (ORB) and servers and clients. ORB communications trace provides hexadecimal representation of the IIOp packets sent and received by processes. It is helpful in diagnosing problems with ORB communications. Knowledge of the IIOp protocol is needed to interpret the trace data.

This property has one of the following values. Each succeeding value increases the amount of information that is captured.

None

Trace data is not recorded for this component.

Basic

The smallest amount of trace information, critical path trace data, is recorded. This data is primarily used for the highest level data and performance measurements.

Intermediate

For ORB communications trace, the data recorded is the same as for the Basic setting.

Advanced

For ORB communications trace, the data recorded is the same as for the Basic setting.

Trace is used by or for IBM Service personnel to assist in collecting data in possible defect situations. This support should only be used under direction of IBM Service personnel. Incorrectly setting trace properties for objects can result in performance degradation for normal operation.

com.ibm.CORBA.orbIRTraceLevel=trace_level

[none] Controls the amount of trace data that is written to the trace log for interface repository (IR) operations performed by the Object Request Broker (ORB).

This property has one of the following values. Each succeeding value increases the amount of information that is captured.

None

Trace data is not recorded for this component.

Basic

The smallest amount of trace information, critical path trace data, is recorded. This data is primarily used for the highest level data and performance measurements.

Intermediate

Record trace messages and any throw instructions that are processed, in addition to the information recorded for the basic trace level.

Advanced

Record all trace information, including process flow and detailed data, in addition to the information recorded for the intermediate trace level. Further, messages sent to the activity and error logs are also recorded in the trace log. This data is primarily for extended problem determination. It controls the recording of extra raw data, component extended messages, and indications that an exception subclass was thrown.

Trace is used by or for IBM Service personnel to assist in collecting data in possible defect situations. This support should only be used under direction of IBM Service personnel. Incorrectly setting trace properties for objects can result in performance degradation for normal operation.

com.ibm.CORBA.orbMutexTraceLevel=*trace_level*

[none] Controls the amount of trace data that is written to the trace log for mutex operations performed by the Object Request Broker (ORB). The ORB mutex trace records data at the base ORB level about threads claiming and releasing mutexes. This can be valuable in debugging deadlock situations.

This property has one of the following values. Each succeeding value increases the amount of information that is captured.

None

Trace data is not recorded for this component.

Basic

The smallest amount of trace information, critical path trace data, is recorded. This data is primarily used for the highest level data and performance measurements.

Intermediate

For ORB mutex trace, the data recorded is the same as for the Basic setting.

Advanced

For ORB mutex trace, the data recorded is the same as for the Basic setting.

Trace is used by or for IBM Service personnel to assist in collecting data in possible defect situations. This support should only be used under direction of IBM Service personnel. Incorrectly setting trace properties for objects can result in performance degradation for normal operation.

com.ibm.CORBA.orbRequestTraceLevel=*trace_level*

[none] Controls the amount of trace data that is written to the trace log for request operations performed by the Object Request Broker (ORB).

This property has one of the following values. Each succeeding value increases the amount of information that is captured.

None

Trace data is not recorded for this component.

Basic

The smallest amount of trace information, critical path trace data, is recorded. This data is primarily used for the highest level data and performance measurements.

Intermediate

Record trace messages and any throw instructions that are processed, in addition to the information recorded for the basic trace level.

Advanced

Record all trace information, including process flow and detailed data, in addition to the information recorded for the intermediate trace level. Further, messages sent to the activity and error logs are also recorded in the trace log. This data is primarily for extended problem determination. It controls the recording of extra raw data, component extended messages, and indications that an exception subclass was thrown.

Trace is used by or for IBM Service personnel to assist in collecting data in possible defect situations. This support should only be used under direction of IBM Service personnel. Incorrectly setting trace properties for objects can result in performance degradation for normal operation.

com.ibm.CORBA.traceLogMaxSize=

[100] The maximum size (in kilobytes) that the trace log can reach. If the trace log reaches this size, it will wrap around, with the oldest messages in the log being overwritten with any new messages. This property is an integer number of kilobytes in the range 100 through infinity. When setting this property make sure that there is enough room available for the trace log on the disk where this directory is located (the directory of the trace log is set by the **com.ibm.CORBA.traceLogDirectory** property).

Trace is used by or for IBM Service personnel to assist in collecting data in possible defect situations. This support should only be used under direction of IBM Service personnel. Incorrectly setting trace properties for objects can result in performance degradation for normal operation.

com.ibm.CORBA.traceLogDirectory=

[] The name of the directory used to store the trace log for the related host. If this value is left blank then the default directory is used. When setting this property make sure that there is enough room available for the trace log on the disk where this directory is located (the maximum size of the trace log is set by the **com.ibm.CORBA.traceLogMaxSize** property).

Trace is used by or for IBM Service personnel to assist in collecting data in possible defect situations. This support should only be used under direction of IBM Service personnel. Incorrectly setting trace properties for objects can result in performance degradation for normal operation.

com.ibm.CORBA.transactionTraceLevel=*trace_level*

[none] Controls the amount of trace data that is written to the trace log for transaction object service operations for servers.

This property has one of the following values. Each succeeding value increases the amount of information that is captured.

None

Trace data is not recorded for this component.

Basic

The smallest amount of trace information, critical path trace data, is recorded. This data is primarily used for the highest level data and performance measurements.

Intermediate

Record trace messages and any throw instructions that are processed, in addition to the information recorded for the basic trace level.

Advanced

Record all trace information, including process flow and detailed data, in addition to the information recorded for the intermediate trace level. Further, messages sent to the activity and error logs are also recorded in the trace log. This data is primarily for extended problem determination. It controls the recording of extra raw data, component extended messages, and indications that an exception subclass was thrown.

Trace is used by or for IBM Service personnel to assist in collecting data in possible defect situations. This support should only be used under direction of IBM Service personnel. Incorrectly setting trace properties for objects can result in performance degradation for normal operation.

Client and server transaction support properties

You can specify the following transaction support properties for clients and servers:

com.ibm.CORBA.defaultTimeout=

[300] The default time, in seconds, after which a top-level transaction is rolled back if it has not completed.

Because a transaction may hold locks on database records, it is important to ensure that all transactions complete within a reasonable period of time. This is especially important in a distributed environment where a transaction may be originated by a non-recoverable client. If such a client dies without ending all the transactions it started, then those transactions should have a period of time after which they are automatically rolled back by the server on which they were created.

This timeout value is the time from when the originator started the top-level transaction to the time when the originator must request that the transaction be committed or rolled back. It is an integer number of seconds greater than 0. If you set this property to 0 (zero), the top-level transaction never times out in the lifetime of the server on which the transaction was created.

If the application server's default transaction timeout is set to 0 (zero), transactions started using the [“CosTransactions::Current interface” on page](#) do not have a time limit set. An application program can set a time limit by calling the `set_timeout()` operation on the `CosTransactions::Current` object, passing the time limit required as a parameter.

com.ibm.CORBA.deferredBegin=*deferbegin*

[always] This property is used to control whether clients should attempt to defer the begin of transactions until the first remote business method is called.

In general, it is desirable for clients to have a transaction created on the same application server as at least one of the Enterprise JavaBean objects. The transaction service provides the ability to defer the creation of a transaction until the first remote business method is called, allowing the transaction service on the remote application server to create the transaction during the processing of that first business method. However, the transaction service on the remote

application server must be capable of supporting this function.

You determine whether clients should attempt to defer the creation of transactions by setting the `com.ibm.CORBA.deferredBegin` property to one of the following values:

always

Always defer the creation of a transaction. This setting can be used even if WebSphere application servers are started with their default property settings. WebSphere application servers handle deferred begins by default, but do not indicate that they support this capability.

never

Never defer the creation of a transaction. When a client application requests that a transaction be begun, a transaction factory is obtained using a factory finder. The factory finder may be specified with the `Factory finder` property. If a value is not specified for the factory finder property, an arbitrary application server is chosen. This setting may be detrimental to performance when communicating with application servers that can support deferred begin, because transactions may be created on an application server that is not otherwise involved in the transaction.

serverDependent

Defer the creation of a transaction depending on the capability of the remote server.

The transaction's client code determines the capability of the remote transaction service to support the deferred begin protocol. The client determines the capability of the remote server from information contained in the target object's proxy object, so no remote flows are required for this test.

WebSphere application servers that are started specifying the property `com.ibm.ejs.jts.jts.ControlSet.nativeOnly=false` export this information in the target object's proxy.

Note: This is not the default startup property for the application server.

com.ibm.CORBA.transactionfactoryFinder=

[(A null value.)] The name to be used to find a transaction factory for transactional clients.

The value is the fully-qualified name path from the local root, which can be used in a resolve to get the factory desired. For example, one possible value to specify is:

```
com.ibm.CORBA.transactionfactoryFinder=node/servers/xyzServer
```

You can specify any transaction factory that is bound into the name space. Through the use of this property, you can direct of the search for a particular transaction factory. The above example finds a factory on the local node (host) in server `xyzServer`. The format of the property value may be either of the following:

```
node/servers/servername  
or  
domain/nodes/nodename/servers/servername
```

where `nodename` is the name of one of the nodes in the configured WebSphere domain and `servername` is the name of a server. If a null value is supplied, a

search starts on the local bootstrap node and if no factory is found, the search when proceeds throughout the domain searching all configured nodes and servers for an available transaction factory. Thus a null default value on a large configuration may incur a performance overhead.

Note: This property is only used if transactions are not deferring the start of a transaction until the first business method.

com.ibm.CORBA.transactionEnabled=

[yes] Whether or not this client is enabled to use the transaction service. The possible values for this property are **yes** or **no**.

Client and server OLT- and debug-specific properties

You can specify the following OLT- and debug-specific properties for clients and servers:

com.ibm.CORBA.debugEnabled=

[yes] Whether or not the OLT runtime is enabled for tracing and debugging. The possible values for this property are **yes** or **no**.

com.ibm.CORBA.oltHostname=*host_name*

The hostname of the machine where the OLT server is running.

com.ibm.CORBA.oltPort=*port_number*

[2102] The number of the port at which the OLT server listens.

For tracing and debugging, the property `com.ibm.CORBA.requestTimeout=0` must be set.

For more information about tracing and debugging, see “4.1.2.1: IBM Distributed Debugger and Object Level Trace” in the WebSphere Application Server Advanced Edition infocenter.

Reference information for problem determination

The following topics provide reference information to help you resolve runtime problems with WebSphere Application Server enterprise services:

- [“Fields in a formatted activity log entry” on page 298](#)
- [“CORBA system exception minor codes” on page 301](#)

Description of a formatted activity log entry

If you use the showlog utility to format an enterprise services' activity log to a file, the first two lines of the output file show the following two fields:

\$LANG

The language of the system where showlog command was run, taken from the system's environment.

\$CODESET

The codepage of the system where showlog command was run, obtained from the function call "char * nl_langinfo(CODESET)".

You cannot change this information which is generated by showlog.

The remainder of the formatted output contains entries for events recorded in the activity log.

Content of a formatted activity log entry

Each entry in the activity log has the following fields:

ComponentId: *number*

A numeric value that identifies the component in WebSphere Application Server enterprise services.

ProcessId: *number*

The process number under which the client or server is running; this is the identifier by which the operating system knows the process.

ThreadId: *number*

The thread identifier under which the object placed the event in the activity log. This is the thread identifier by which the operating system knows the thread.

FunctionName: *name*

An internal name for the function that placed the information in the activity log. This may not be very useful for you.

Probeld: *number*

This is typically the line number in the source file that has the function that placed the entry in the event log. However, some components use this field as a probe identifier and assign it some other value.

Sourceld: *number filename*

The number identifies the version of code that is running; the second value is the location of the source code file in the library system from which the code was built. This information may not be very useful to you.

Manufacturer: IBM

Product: WebSphere Application Server

Version: *number*

The version of the product.

SOMProcessType: *number*

The type of process that placed this event in the activity log:

1

A client process.

2

An ORB daemon.

5

A server process.

ServerName: *name*

If this is a server process, the name of the server.

ClientHostName: *hostname*

The host name of the client associated with the event, if security is enabled for the server process.

ClientUserId: *userid*

The user ID of the client associated with the event, if security is enabled for the server process.

TimeStamp: *yyyy-mm-dd hh:mm:ss.nnnnnnnnn*

The date and time when the event was placed in the activity log, in the format: *yyyy-mm-dd hh:mm:ss.nnnnnnnnn* , where:

yyyy-mm-dd

The date of the event, as a numeric form of year, month, and day (ISO 8601 standard format); for example, 2001-08-15 for the 15th August 2001.

hh:mm:ss.nnnnnnnnn

The time of the event, as a numeric form of hour, minute, and second; for example, 19:45:01.149182980.

UnitOfWork: *nnnnn:hhhhh*

The unit-of-work identifier for the original request, in the format, *nnnnn:hhhhh*, where:

nnnnn

A random number.

hhhhh

The name of the host where the original request originated.

If a request is forwarded to another server as part of the original request, the unit of work from the original request is used. This enables clients to track the work that is done as part of an original request.

Note: You may find the UnitOfWork information useful when you are trying to find related entries in the activity log or when you are debugging problems across multiple machines. For more information about how to view activity log entries in UnitOfWork sequence, see [“Reading the activity log” on page .](#)

Severity: *number*

The severity of the problem. The possible values are:

1

Error - Indicates there is a problem with the operation that caused the message, and the operation has failed. In this case, see other messages in the log with the

same UnitOfWork for additional information.

2

Warning - Indicates there may be a problem with the operation that caused this message. The operation continues and may or may not be successful. In this case, see other messages in the log with the same UnitOfWork for additional information.

3

Informational - The message is informational only. The operation that caused this message continues. In this case, see other messages in the log with the same UnitOfWork for additional information.

Category: *number*

The category of the failure.

For messages generated from C++, the possible values are:

1

Error - Indicates that a severe error has occurred. These messages are written to the error log, activity log and the Windows platforms event log, which can be viewed using Windows Event Viewer.

2

Activity - Indicates some general form of activity. These messages have one of the three **Severity** settings and are written to the activity log.

3

Trace - Is used for component trace. These messages are written to trace logs.

4

Trace Data - Is used for additional data for component trace. These messages are written to trace logs.

5

Performance - Is used for performance trace data. These messages are written to trace logs.

For messages generated from C++, the possible values are:

AUDIT

An informational message, written to the activity log.

WARNING

A serious warning message, written to the activity log.

ERROR

A severe error, written to the error log, activity log, and the Windows platforms event log, which can be viewed using Windows Event Viewer.

EVENT, ENTRY, EXIT, DEBUG

Trace messages, written to the trace log.

FormatWarning: *number*

A non-zero numeric value when an attempt to place the replacement text in the Primary or Extended Message was not correct.

PrimaryMessage:

In the case that the entry was placed in the activity log as part of an exception, the PrimaryMessage field contains the `objectName::methodName(parameter`

`list):lineNumber` and the type of the exception (this should always be `CORBA::exception`) and the specific exception that was raised.

This field contains essential information for problem determination and indicates one of the following:

Throw of exception

The entry indicates that the listed function determined that an exception should be thrown. Look at the **ExtendedMessage** information to help you identify the cause of the problem. This is a category 1 (Error) entry.

Reraised exception

The entry indicates that an exception was received and reraised. This entry allows you to trace related entries in the activity log. This is a category 1 (Error) entry.

Mapping of exception

The entry shows the exception that was received and the new exception that was raised. This is useful when tracking a specific exception through the activity log. This is a category 1 (Error) entry.

Activity entry

An activity entry provides information. If the entry appears during an exception path, it contains information to help determine the cause of the problem for which the exception is being raised. If the entry is not on an exception path, the entry provides information as to the state of the server. This is a category 2 (Activity) entry.

Note: If you see a minor code of this format, 0x49420xxx, where xxx are hexadecimal numbers, see [“CORBA system exception minor codes” on page 30](#) for more information about the message.

ExtendedMessage:

The extended message often provides additional information to pinpoint the exact cause of the failure.

RawDataLen:

When raw data is provided as part of this log entry, the length of the raw data is shown in hexadecimal format. The raw data follows this entry and is shown in a 16-byte dump with the ASCII format of the data at the right of the dump.

If the ORB communications trace is turned on, the trace data from GIOP packets is displayed in the RawData field.

CORBA system exception minor codes

In the CORBA model for exception handling, all exceptions can be associated with minor codes. This topic provides details of these minor codes, in hexadecimal order.

Minor codes are used in several ways:

- They are returned in the minor code field of exception bodies (when appropriate).
- They are placed in the activity log as part of the PrimaryMessage.
- They can be written in diagnostic messages on a computer screen.

There is not a one-to-one mapping of system exceptions to minor codes. A single minor code can be associated with several different exceptions, and the diagnostic message can be different depending on which exception was thrown.

Each minor code is a three digit hexadecimal serial number. This number is prefixed with 0x49420, which is the OMG-assigned vendor identification code for the C++ ORB provided with IBM Websphere Application Server Enterprise Services.

Note: In some cases, minor code numbers may be reported without the vendor ID. Minor codes reported from Java are in decimal point and lack the vendor ID.

Minor code numbers are unique within the scope for each system exception, but there is no restriction that minor code numbers be unique across all system exceptions.

In this topic, the description of each minor code consists of:

Minor Code number (prefixed with vendor ID) : Error Text (the text string that identifies the minor code)

Explanation: A description of the problem that caused the error.

User Response: Actions needed to resolve the problem, if appropriate.

Minor code definitions

0x49420032 : SOMDERROR_CouldNotLoadLibrary

Explanation: Client initialization cannot load a required library.

User Response: Check the log for more information.

0x49420033 : SOMDERROR_NoMemory

Explanation: A memory allocation failed.

User Response: Verify that process does not have a memory leak. Increase system resources.

0x49420034 : SOMDERROR_NotImplemented

Explanation: The invoked operation is not supported in the product or is not valid on the target object

User Response: Check that the operation being invoked and the target object run-time type are compatible. Refer to the documentation for the operation for information about restrictions.

0x49420035 : SOMDERROR_InvalidProtocolInformation

Explanation: The configuration of the communications protocol is incorrect. Supported communication protocols are TCP/IP and IPC.

User Response: Ensure that at least one valid communications protocol image was configured using system management (for either TCP/IP or IPC). Ensure that a host image was configured using the correct host name. Ensure that for each communications protocol configured, the csProfileTag and portNumber are set and that the portNumber is not using another process on the system. (The portNumber is the port on which the location service daemon listens for requests.) The csProfileTag and portNumber settings must be unique for each communications protocol. Ensure that for each server registered in the Implementation Repository, the set of supported communication protocols intersects with the set of communications protocol images configured using system management.

0x49420036 : SOMDERROR_SOMDDAAlreadyRunning

Explanation: The location service daemon cannot begin listening because another process is using the port number. Probably another instance of the process is already running.

User Response: Do not attempt to start the location service daemon or terminate the other instance. If no other location service daemon is running, try reconfiguring

the location service daemon to listen on a different port number. Each communications protocol is configured with a separate port number using system management.

0x49420037 : SOMDERROR_InvalidConfigSetting

Explanation: A configuration setting or environment variable was not properly set.

User Response: An error log entry indicates which configuration setting or environment variable is not properly set. If the reported variable is HOSTNAME, ensure that a host image was configured using system management. If the reported variable is SOMCBASE, ensure that the product was properly installed (SOMCBASE should be set to the directory where the product was installed.) If the reported variable is SOMCBENV, ensure that SOMCBENV has one of the following forms: D:<image-name> S:<image-name> C:<image-name> A:<image-name> where "D:" starts the location-service daemon, "S:" starts a server process, "C:" starts a client process, and "A:" starts a systems management agent process. The <image-name> is the name of a systems management image. For servers, the image name is the same as the server alias. For non-managed clients, the SOMCBENV environment variable should be set to the name of a configuration file that contains configuration settings for the process. The default configuration file somcbenv.ini is in the "etc" subdirectory of the installed product directory.

0x49420038 : SOMDERROR_HostAddress

Explanation: Cannot map a host name on a different machine to a host address.

User Response: Ensure that the host with which this process is attempting to communicate is known and can be reached via TCP/IP. Try to ping the remote host by the host name.

0x49420039 : SOMDERROR_CouldNotStartProcess

Explanation: The location daemon cannot start a server process.

User Response: Check the log for more information.

0x4942003A : SOMDERROR_CouldNotStartThread

Explanation: Cannot start a thread.

User Response: Check the log for more information. Increase system resources.

0x4942003B : SOMDERROR_NoMessages

Explanation: No request messages were pending in a server process when the server invoked CORBA::BOA::execute_next_request or CORBA::BOA::execute_request_loop with the CORBA::BOA::SOMD_NO_WAIT flag.

User Response: Wait for a request to become available, or use the CORBA::BOA::SOMD_WAIT flag to call CORBA::BOA::execute_next_request or CORBA::BOA::execute_request_loop.

0x4942003C : SOMDERROR_MarshalingError

Explanation: An error has occurred when trying to marshall or demarshall method parameters or return results as part of a remote invocation. This can occur if the process attempts to pass a IOM proxy as a method parameter or return result. Only objects that inherit from CORBA::Object_ORBProxy can be passed on cross-process invocations. It can also occur when demarshalling an inout sequence if the length of the incoming sequence is greater than the original sequence maximum. It can occur if methods are not invoked on the ServerRequest object in the correct order when using the Dynamic Skeleton Interface (DSI).

User Response: Ensure that IOM proxies are not passed as method parameters or return results. Ensure that inout sequences do not exceed the sequence maximum. If using the DSI, ensure that operations are invoked in the correct order on the

ServerRequest object.

0x4942003D : SOMDERROR_CommTimeOut

Explanation: A process has timed out while waiting for a response from another process. Typically a client receives this error when the server has terminated or is hanging due to an application error.

User Response: Ensure that the other process is still active. To increase the timeout period, change the request timeout setting using system management.

Note: Setting the request timeout setting to zero results in an infinite timeout.

0x4942003E : SOMDERROR_CannotConnect

Explanation: A client process cannot connect to a server process when attempting to invoke a method on a proxy to an object residing in that server process.

User Response: Ensure that the location service daemon is running on the machine on which the server resides. Ensure that the object reference is still valid. Try to ping the remote machine to see that the two machines are connected.

0x4942003F : SOMDERROR_No_Server_Available

Explanation: A client has invoked a method on a proxy to an object residing in a server group, but no server in the server group is currently available or the server group cannot be reached. Either a method call was made on a server group aware object, but the server group has no servers configured in it. Or a method call was made on a server group aware object for which there is at least one server configured, but none of the servers that are available were selected by the configured bind policies. Can be the result of a permanent server failure or communications failure.

User Response: Configure at least one server into the server group if the server group has no configured servers. If the server group has at least one server configured, then ensure that the configured bind policies are deselecting all the available servers or the bind policy may have to be modified. Alternatively, there may be a problem communicating with one or more of the servers. Ensure that the servers in the server group are running and that there is communication between the client or server and the servers in the server group. Shut down and restart the client or server and reinitialize the application that caused the error. Alternatively, catch the error and retry the method call until a server becomes available.

0x49420040 : SOMDERROR_BadObjref

Explanation: An invalid object reference was used. For example, if the server receives a reference to an object that no longer exists or cannot be located in that server, this error is sent from a server to a client. This error can occur in a client process if an invalid string is passed to CORBA::ORB::string_to_object. This error occurs in a server if CORBA::BOA::create is called with input ReferenceData that doesn't map to any known exportable object residing in that server. The error occurs if CORBA::BOA::get_id is invoked on a nil object reference or on an object reference that has no associated ReferenceData in that server. Also, this error occurs if a server attempts to export an object reference that has no associated ReferenceData in that server, or if a non-server attempts to pass a local object as a parameter on a remote method invocation. (A " non-server" is any process that has not yet called CORBA::BOA::impl_is_ready.)

User Response: In a client process, ensure that the object which the object reference refers to still exists. Ensure that strings passed to CORBA::ORB::string_to_object have not been corrupted or truncated. There is no maximum length for an object reference string; some are larger than others. Ensure that servers do not attempt to export objects that are not handled by the application

adaptor of the server. IOM proxies cannot be exported from a server.

0x49420041 : SOMDERROR_Unknown

Explanation: An unexpected error occurred during an operation.

User Response: Report the occurrence to technical support.

0x49420042 : SOMDERROR_CommunicationsError

Explanation: A communications failure occurred. Possible reasons are:

- A process could have received an unknown or unexpected message type or message content
- The process could have encountered a low-level communications failure in attempting to send a message or binding to a socket
- An unexpected broken connection could have occurred.

User Response: Ensure that communications resources are functioning properly; for example, when using TCP/IP, try to ping the remote host. Ensure that the process has not failed due to an application error.

0x49420043 : SOMDERROR_ImplRepoI

Explanation: Cannot access the Implementation Repository database.

User Response: Ensure that the Implementation Repository was correctly created and configured using system management. Each host machine must have its own Implementation Repository.

0x49420044 : SOMDERROR_EntryNotFound

Explanation: Cannot find an entry in the Implementation Repository when attempting to delete, update, or locate it.

User Response: Ensure that the specified server alias or UUID matches a server that was previously registered in the Implementation Repository.

0x49420045 : SOMDERROR_ClassNotFound

Explanation: Cannot convert an IOR to an object. The class name was unknown or the proxy factory cannot be created.

User Response: Verify the class implementation and make sure that the bindings exist.

0x49420046 : SOMDERROR_ServerAlreadyExists

Explanation: A server cannot register with the location service daemon during CORBA::BOA::impl_is_ready. Another server may already be registered with the location service daemon under the server UUID. Only one instance of a particular server can be running on a given host.

User Response: Terminate the duplicate server process. If no duplicate server process is running, restart the location service daemon.

0x49420047 : SOMDERROR_CtxNoPropFound

Explanation: Cannot find a specified CORBA::Context property. This error occurs if an invalid property name was passed to CORBA::Context::delete_values.

User Response: Ensure that the specified property name exists in the context object

0x49420048 : SOMDERROR_BadParm

Explanation: An application supplied an invalid parameter to an operation.

User Response: Check the error log for a message indicating which operation was given the invalid parameter. Check the documentation for that operation and ensure that the passed parameters are valid.

0x49420049 : SOMDERROR_AuthnFail

Explanation: An application attempted to manipulate an entry in the Implementation Repository for a server that is either being managed or disabled by system management. Only entries registered using the ImplRepository interface can be updated or deleted using the ImplRepository interface. Such entries in the Implementation Repository cannot be deleted or updated using the ImplRepository programmatic interface. The error is also raised if CORBA::ImplRepository::find_impldef is used to find a server that was disabled by system management.

User Response: Manipulate the server using system management. Ensure that the server was not disabled by system management. Ensure that all entries in the Implementation Repository that are to be deleted or updated, were originally added programmatically and not by using system management.

0x4942004A : SOMDERROR_DuplicateEntry

Explanation: The application attempted to add a duplicate entry to the Implementation Repository, or attempted to update the server alias of an existing entry using a name that is not unique. The server alias need not be unique throughout the network but must be unique in each Implementation Repository.

User Response: Ensure that the server UUID and server alias of the ImplementationDef to be added or updated in the Implementation Repository are unique.

0x4942004B : SOMDERROR_Internal

Explanation: Unknown.

User Response: Report the occurrence to technical support.

0x4942004D : SOMDERROR_WrongRefType

Explanation: The wrong type of object reference was used. Probably, a client invoked an operation on an object in a server and the object did not support the invoked method. To support a given operation, a server must have been compiled and linked with the server-side C++ bindings for the interface that introduces that IDL operation. This error also occurs when a server application invokes CORBA::BOA::get_id and passes in a proxy object rather than a local object.

User Response: Ensure that a server is compiled and linked with all the server-side C++ bindings for the interfaces it exports. Ensure that a server does not pass a proxy object to CORBA::BOA::get_id.

0x4942004E : SOMDERROR_SOMDDNotRunning

Explanation: A server cannot register with the location service daemon (in CORBA::BOA::impl_is_ready), because it cannot contact the daemon. Maybe the daemon not running, or the daemon running on a port number that is different from what the server expected.

User Response: Ensure that the location service daemon is running on the same host as the server. Ensure that the port number configuration setting for each communications protocol is the same for the systems management server image and daemon image.

0x49420051 : SOMDERROR_DataConversion

Explanation: Cannot perform code set translation for character data. This results from a failure of the XPG4 functions iconv() or nl_langinfo(). It can occur if the process is using a non-standard XPG4 code set that does not map to an OSF code set. It can occur if the native code set for the process (as reported by the XPG4 function nl_langinfo) does not match the nativeCharSet configuration data of the process (which was configured using system management). It can occur if a server

does not have XPG4 code set converters for the transmission code set chosen by the client process. It can occur if the char code sets configuration setting for the server contains one or more code sets for which the process cannot open (using iconv_open) XPG4 converters. It can occur if there is no common code set between the client and the server.

User Response: When using the translationEnabled configuration setting, ensure that the NLS-related configuration settings have been correctly set. Also ensure that the correct XPG4 code set converters have been installed and that all environment variables (such as LOCPATH) required by XPG4 have been properly set. Ensure that both the client and the server are using standard code sets and that there is some code set supported by both the client and the server.

0x49420052 : SOMDERROR_IRIncoherent

Explanation: An Interface Repository object references another named Interface Repository object which no longer resides in the IR database.

User Response: Contact IBM Support and report the problem.

0x49420053 : SOMDERROR_IRInternal

Explanation: An internal programming or database error has occurred.

User Response: Contact IBM Support and report the problem.

0x49420054 : SOMDERROR_IRDuplicateEntry

Explanation: Attempted to create an Interface Repository object where one already exists in the Interface Repository with either the same CORBA::RepositoryId or the same name within that container.

User Response: Change the ID (CORBA::RepositoryId) parameter that is passed to the 'create_xxxx' operation, or change the ID (CORBA::RepositoryId) value of the object already in the IR which is causing the duplicate entry error via the ID write operation. Change the name of one of the two conflicting objects within that container.

0x49420055 : SOMDERROR_IReEntryNotFound

Explanation: One of the input parameters of a create_xxxx operation referenced an Interface Repository object which is not in the database.

User Response: Specify a named object that exists in the Interface Repository database.

0x49420056 : SOMDERROR_IRCannotConnect

Explanation: Cannot find or access the Interface Repository database. This occurs during a call to resolve_initial_references (with an input string of InterfaceRepository).

User Response: Ensure that the Interface Repository database exists and is properly configured. Ensure that the directory or file permissions associated with the Interface Repository database allow access by the user receiving the exception.

0x49420057 : SOMDERROR_IRInUse

Explanation: Another thread or process is updating that portion of the Interface Repository database.

User Response: Retry the Interface Repository operation that generated the exception at a later time.

Notices

This information was developed for products and services offered in the U.S.A. IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:
IBM Director of Licensing IBM Corporation North Castle Drive Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:
IBM World Trade Asia Corporation Licensing 2-31 Roppongi 3-chome, Minato-ku Tokyo 106,
Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS DOCUMENT "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OR CONDITIONS OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the document. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:
IBM Corporation Department LZKS 11400 Burnet Road Austin, TX 78758 U.S.A.

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

The licensed program described in this document and all licensed material available for it

are provided by IBM under terms of the IBM International Program License Agreement or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

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

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

Trademarks and service marks

The following terms are trademarks or registered trademarks of the IBM Corporation in the United States, other countries, or both:

| | | |
|---|-----|--|
| Advanced Peer-to-Peer Networking AFS AIX APPN AS/400 CICS CICS OS/2 CICS/400 CICS/6000 CICS/ESA CICS/MVS CICS/VSE CICSplex DB2 DB2 Universal Database DCE Encina Lightweight Client DFS Encina IBM IBM System Application Architecture IMS IMS/ESA Language Environment | *** | MQSeries MVS/ESA NetView Open Class OS/2 OS/390 OS/400 Parallel Sysplex PowerPC RACF RAMAO RMF RISC System/6000 RS/6000 S/390 SAA SecureWay TeamConnection Transarc TXSeries VSE/ESA VTAM VisualAge WebSphere |
|---|-----|--|

Domino, Lotus, and LotusScript are trademarks or registered trademarks of Lotus Development Corporation in the United States, other countries, or both.

Tivoli is a registered trademark of Tivoli Systems, Inc. in the United States, other countries, or both.

ActiveX, Microsoft, Visual Basic, Visual C++, Visual J++, Windows, Windows NT, and the Windows 95 logo are trademarks or registered trademarks of Microsoft Corporation in the United States, other countries, or both.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Some of this documentation is based on material from Object Management Group bearing the following copyright notices:

Copyright 1995, 1996 AT&T/NCR
Copyright 1995, 1996 BNR Europe Ltd.
Copyright 1991, 1992, 1995, 1996 by Digital Equipment Corporation
Copyright 1996 Gradient Technologies, Inc.
Copyright 1995, 1996 Groupe Bull
Copyright 1995, 1996 Expersoft Corporation
Copyright 1996 FUJITSU LIMITED
Copyright 1996 Genesis Development Corporation
Copyright 1989, 1990, 1991, 1992, 1995, 1996 by Hewlett-Packard Company
Copyright 1991, 1992, 1995, 1996 by HyperDesk Corporation
Copyright 1995, 1996 IBM Corporation
Copyright 1995, 1996 ICL, plc
Copyright 1995, 1996 Ing. C. Olivetti &C.Sp
Copyright 1997 International Computers Limited
Copyright 1995, 1996 IONA Technologies, Ltd.
Copyright 1995, 1996 Itasca Systems, Inc.
Copyright 1991, 1992, 1995, 1996 by NCR Corporation
Copyright 1997 Netscape Communications Corporation
Copyright 1997 Northern Telecom Limited
Copyright 1995, 1996 Novell USG
Copyright 1995, 1996 02 Technologies

Copyright 1991, 1992, 1995, 1996 by Object Design, Inc.
Copyright 1991, 1992, 1995, 1996 Object Management Group, Inc.
Copyright 1995, 1996 Objectivity, Inc.
Copyright 1995, 1996 Oracle Corporation
Copyright 1995, 1996 Persistence Software
Copyright 1995, 1996 Servio, Corp.
Copyright 1996 Siemens Nixdorf Informationssysteme AG
Copyright 1991, 1992, 1995, 1996 by Sun Microsystems, Inc.
Copyright 1995, 1996 SunSoft, Inc.
Copyright 1996 Sybase, Inc.
Copyright 1996 Taligent, Inc.
Copyright 1995, 1996 Tandem Computers, Inc.
Copyright 1995, 1996 Teknekron Software Systems, Inc.
Copyright 1995, 1996 Tivoli Systems, Inc.
Copyright 1995, 1996 Transarc Corporation
Copyright 1995, 1996 Versant Object Technology Corporation
Copyright 1997 Visigenic Software, Inc.
Copyright 1996 Visual Edge Software, Ltd.

Each of the copyright holders listed above has agreed that no person shall be deemed to have infringed the copyright in the included material of any such copyright holder by reason of having used the specification set forth herein or having conformed any computer software to the specification.

WHILE THE INFORMATION IN THIS PUBLICATION IS BELIEVED TO BE ACCURATE, THE OBJECT MANAGEMENT GROUP, AND THE COMPANIES LISTED ABOVE MAKE NO WARRANTY OF ANY KIND WITH REGARDS TO THIS MATERIAL INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. The Object Management Group and the companies listed above shall not be liable for errors contained herein or for incidental or consequential damages in connection with the furnishing, performance, or use of this material.



This software contains RSA encryption code.

Other company, product, and service names may be trademarks or service marks of others.