OMG IDL Syntax and Semantics 3

This chapter has been updated based on CORE changes from ptc/98-09-04 and the Objects by Value documents (ptc/98-07-05 and orbos/98-01-18). This chapter describes OMG Interface Definition Language (IDL) semantics and gives the syntax for OMG IDL grammatical constructs.

Contents

This chapter contains the following sections.

3.1 Overview

The OMG Interface Definition Language (IDL) is the language used to describe the interfaces that client objects call and object implementations provide. An interface definition written in OMG IDL completely defines the interface and fully specifies each operation’s parameters. An OMG IDL interface provides the information needed to develop clients that use the interface’s operations.

Clients are not written in OMG IDL, which is purely a descriptive language, but in languages for which mappings from OMG IDL concepts have been defined. The mapping of an OMG IDL concept to a client language construct will depend on the facilities available in the client language. For example, an OMG IDL exception might be mapped to a structure in a language that has no notion of exception, or to an exception in a language that does. The binding of OMG IDL concepts to several programming languages is described in this manual.

OMG IDL obeys the same lexical rules as C++[1], although new keywords are introduced to support distribution concepts. It also provides full support for standard C++ preprocessing features. The OMG IDL specification is expected to track relevant changes to C++ introduced by the ANSI standardization effort.

The description of OMG IDL’s lexical conventions is presented in Section 3.2, “Lexical Conventions,” on page 3-3. A description of OMG IDL preprocessing is presented in Section 3.3, “Preprocessing,” on page 3-12. The scope rules for identifiers in an OMG IDL specification are described in Section 3.14, “CORBA Module,” on page 3-44.

The OMG IDL grammar is a subset of the proposed ANSI C++ standard, with additional constructs to support the operation invocation mechanism. OMG IDL is a declarative language. It supports C++ syntax for constant, type, and operation declarations; it does not include any algorithmic structures or variables. The grammar is presented in Section 3.4, “OMG IDL Grammar,” on page 3-12.

OMG IDL-specific pragmas (those not defined for C++) may appear anywhere in a specification; the textual location of these pragmas may be semantically constrained by a particular implementation.

1. Ellis, Margaret A. and Bjarne Stroustrup, *The Annotated C++ Reference Manual*, Addison-Wesley Publishing Company, Reading, Massachusetts, 1990, ISBN 0-201-51459-1

A source file containing interface specifications written in OMG IDL must have an ".idl" extension. The file orb.idl contains OMG IDL type definitions and is available on every ORB implementation.

The description of OMG IDL grammar uses a syntax notation that is similar to Extended Backus-Naur Format (EBNF). Table 3-1 lists the symbols used in this format and their meaning.

Table 3-1 IDL EBNF

Symbol	Meaning
::=	Is defined to be
|	Alternatively
<text>	Nonterminal
"text"	Literal
*	The preceding syntactic unit can be repeated zero or more times
+	The preceding syntactic unit can be repeated one or more times
{}	The enclosed syntactic units are grouped as a single syntactic unit
[]	The enclosed syntactic unit is optional—may occur zero or one time

3.2 Lexical Conventions

This section[2] presents the lexical conventions of OMG IDL. It defines tokens in an OMG IDL specification and describes comments, identifiers, keywords, and literals—integer, character, and floating point constants and string literals.

An OMG IDL specification logically consists of one or more files. A file is conceptually translated in several phases.

The first phase is preprocessing, which performs file inclusion and macro substitution. Preprocessing is controlled by directives introduced by lines having # as the first character other than white space. The result of preprocessing is a sequence of tokens. Such a sequence of tokens, that is, a file after preprocessing, is called a translation unit.

OMG IDL uses the ASCII character set, except for string literals and character literals, which use the ISO Latin-1 (8859.1) character set. The ISO Latin-1 character set is divided into alphabetic characters (letters) digits, graphic characters, the space (blank)

2. This section is an adaptation of *The Annotated C++ Reference Manual*, Chapter 2; it differs in the list of legal keywords and punctuation.

character, and formatting characters. Table 3-2 shows the ISO Latin-1 alphabetic characters; upper and lower case equivalences are paired. The ASCII alphabetic characters are shown in the left-hand column of Table 3-2.

Table 3-2 The 114 Alphabetic Characters (Letters)

Char.	Description	Char.	Description
Aa	Upper/Lower-case A	Àà	Upper/Lower-case A with grave accent
Bb	Upper/Lower-case B	Áá	Upper/Lower-case A with acute accent
Cc	Upper/Lower-case C	Ââ	Upper/Lower-case A with circumflex accent
Dd	Upper/Lower-case D	Ãã	Upper/Lower-case A with tilde
Ee	Upper/Lower-case E	Ää	Upper/Lower-case A with diaeresis
Ff	Upper/Lower-case F	Åå	Upper/Lower-case A with ring above
Gg	Upper/Lower-case G	Ææ	Upper/Lower-case dipthong A with E
Hh	Upper/Lower-case H	Çç	Upper/Lower-case C with cedilla
Ii	Upper/Lower-case I	Èè	Upper/Lower-case E with grave accent
Jj	Upper/Lower-case J	Éé	Upper/Lower-case E with acute accent
Kk	Upper/Lower-case K	Êê	Upper/Lower-case E with circumflex accent
Ll	Upper/Lower-case L	Ëë	Upper/Lower-case E with diaeresis
Mm	Upper/Lower-case M	Ìì	Upper/Lower-case I with grave accent
Nn	Upper/Lower-case N	Íí	Upper/Lower-case I with acute accent
Oo	Upper/Lower-case O	Îî	Upper/Lower-case I with circumflex accent
Pp	Upper/Lower-case P	Ïï	Upper/Lower-case I with diaeresis
Qq	Upper/Lower-case Q	Ññ	Upper/Lower-case N with tilde
Rr	Upper/Lower-case R	Òò	Upper/Lower-case O with grave accent
Ss	Upper/Lower-case S	Óó	Upper/Lower-case O with acute accent
Tt	Upper/Lower-case T	Ôô	Upper/Lower-case O with circumflex accent
Uu	Upper/Lower-case U	Õõ	Upper/Lower-case O with tilde
Vv	Upper/Lower-case V	Öö	Upper/Lower-case O with diaeresis
Ww	Upper/Lower-case W	Øø	Upper/Lower-case O with oblique stroke
Xx	Upper/Lower-case X	Ùù	Upper/Lower-case U with grave accent
Yy	Upper/Lower-case Y	Úú	Upper/Lower-case U with acute accent
Zz	Upper/Lower-case Z	Ûû	Upper/Lower-case U with circumflex accent
		Üü	Upper/Lower-case U with diaeresis
		ß	Lower-case German sharp S
		ÿ	Lower-case Y with diaeresis

Table 3-3 lists the decimal digit characters.

Table 3-3 Decimal Digits

0 1 2 3 4 5 6 7 8 9

Table 3-4 shows the graphic characters.

Table 3-4 The 65 Graphic Characters

Char.	Description	Char.	Description
!	exclamation point	¡	inverted exclamation mark
"	double quote	¢	cent sign
#	number sign	£	pound sign
$	dollar sign	¤	currency sign
%	percent sign	¥	yen sign
&	ampersand	¦	broken bar
’	apostrophe	§	section/paragraph sign
(	left parenthesis	¨	diaeresis
)	right parenthesis	©	copyright sign
*	asterisk	ª	feminine ordinal indicator
+	plus sign	«	left angle quotation mark
,	comma	¬	not sign
-	hyphen, minus sign		soft hyphen
.	period, full stop	®	registered trade mark sign
/	solidus	¯	macron
:	colon	°	ring above, degree sign
;	semicolon	±	plus-minus sign
<	less-than sign	²	superscript two
=	equals sign	³	superscript three
>	greater-than sign	´	acute
?	question mark	µ	micro
@	commercial at	¶	pilcrow
[	left square bracket	·	middle dot
\	reverse solidus	¸	cedilla
]	right square bracket	¹	superscript one
^	circumflex	º	masculine ordinal indicator
_	low line, underscore	»	right angle quotation mark
‘	grave	¼	vulgar fraction 1/4
{	left curly bracket	½	vulgar fraction 1/2
|	vertical line	¾	vulgar fraction 3/4
}	right curly bracket	¿	inverted question mark
~	tilde	×	multiplication sign
		÷	division sign

The formatting characters are shown in Table 3-5.

Table 3-5 The Formatting Characters

Description	Abbreviation	ISO 646 Octal Value
alert	BEL	007
backspace	BS	010
horizontal tab	HT	011
newline	NL, LF	012
vertical tab	VT	013
form feed	FF	014
carriage return	CR	015

3.2.1 Tokens

There are five kinds of tokens: identifiers, keywords, literals, operators, and other separators. Blanks, horizontal and vertical tabs, newlines, formfeeds, and comments (collective, “white space”), as described below, are ignored except as they serve to separate tokens. Some white space is required to separate otherwise adjacent identifiers, keywords, and constants.

If the input stream has been parsed into tokens up to a given character, the next token is taken to be the longest string of characters that could possibly constitute a token.

3.2.2 Comments

The characters /* start a comment, which terminates with the characters */. These comments do not nest. The characters // start a comment, which terminates at the end of the line on which they occur. The comment characters //, /*, and */ have no special meaning within a // comment and are treated just like other characters. Similarly, the comment characters // and /* have no special meaning within a /* comment. Comments may contain alphabetic, digit, graphic, space, horizontal tab, vertical tab, form feed, and newline characters.

3.2.3 Identifiers

An identifier is an arbitrarily long sequence of ASCII alphabetic, digit, and underscore (“_”) characters. The first character must be an ASCII alphabetic character. All characters are significant.

When comparing two identifiers to see if they collide:

- Upper- and lower-case letters are treated as the same letter. Table 3-2 on page 3-4 defines the equivalence mapping of upper- and lower-case letters.
- All characters are significant.

Identifiers that differ only in case collide, and will yield a compilation error under certain circumstances. An identifier for a given definition must be spelled identically (e.g., with respect to case) throughout a specification.

There is only one namespace for OMG IDL identifiers in each scope. Using the same identifier for a constant and an interface, for example, produces a compilation error.

For example:

```
module M {
    typedef long Foo;
    const long thing = 1;
    interface thing {          // error: reuse of identifier
        void doit (
            in Foo foo         // error: Foo and foo collide and refer to
                                   different things
        );

        readonly attribute long Attribute; // error: Attribute collides with
                                                keyword attribute
    };
};
```

3.2.3.1 *Escaped Identifiers*

As IDL evolves, new keywords that are added to the IDL language may inadvertently collide with identifiers used in existing IDL and programs that use that IDL. Fixing these collisions will require not only the IDL to be modified, but programming language code that depends upon that IDL will have to change as well. The language mapping rules for the renamed IDL identifiers will cause the mapped identifier names (e.g., method names) to be changed.

To minimize the amount of work, users may lexically “escape” identifiers by prepending an underscore (_) to an identifier. This is a purely lexical convention which ONLY turns off keyword checking. The resulting identifier follows all the other rules for identifier processing. For example, the identifier _**AnIdentifier** is treated as if it were **AnIdentifier**.

The following is a non-exclusive list of implications of these rules:

- The underscore does not appear in the Interface Repository.
- The underscore is not used in the DII and DSI.
- The underscore is not transmitted over “the wire”.
- Case sensitivity rules are applied to the identifier after stripping off the leading underscore.

For example:

```
module M {
    interface thing {
        attribute boolean abstract;      // error: abstract collides with
                                         //    keyword abstract
        attribute boolean _abstract;     // ok: abstract is an identifier
    };
};
```

To avoid unnecessary confusion for readers of IDL, it is recommended that interfaces only use the escaped form of identifiers when the unescaped form clashes with a newly introduced IDL keyword. It is also recommended that interface designers avoid defining new identifiers that are known to require escaping. Escaped literals are only recommended for IDL that expresses legacy interface, or for IDL that is mechanically generated.

3.2.4 Keywords

The identifiers listed in Table 3-6 are reserved for use as keywords and may not be used otherwise, unless escaped with a leading underscore**.**

Table 3-6 Keywords

abstract	double	long	readonly	unsigned
any	enum	module	sequence	union
attribute	exception	native	short	ValueBase
boolean	factory	Object	string	valuetype
case	FALSE	octet	struct	void
char	fixed	oneway	supports	wchar
const	float	out	switch	wstring
context	in	private	TRUE	
custom	inout	public	truncatable	
default	interface	raises	typedef	

Keywords must be written exactly as shown in the above list. Identifiers that collide with keywords (see Section 3.2.3, “Identifiers,” on page 3-6) are illegal. For example, “**boolean**” is a valid keyword; “**Boolean**” and “**BOOLEAN**” are illegal identifiers.

For example:

```
module M {
    typedef Long Foo;                // Error: keyword is long not Long
    typedef boolean BOOLEAN;         // Error: BOOLEAN collides with
                                     // the keyword boolean;
};
```

OMG IDL specifications use the characters shown in Table 3-7 as punctuation.

Table 3-7 Punctuation Characters

;	{	}	:	,	=	+	-	(	)	<	>	[	]
'	"	\	|	^	&	*	/	%	~				

In addition, the tokens listed in Table 3-8 are used by the preprocessor.

Table 3-8 Preprocessor Tokens

#	##	!	||	&&

3.2.5 Literals

This section describes the following literals:

- Integer
- Character
- Floating-point
- String
- Fixed-point

3.2.5.1 Integer Literals

An integer literal consisting of a sequence of digits is taken to be decimal (base ten) unless it begins with 0 (digit zero). A sequence of digits starting with 0 is taken to be an octal integer (base eight). The digits 8 and 9 are not octal digits. A sequence of digits preceded by 0x or 0X is taken to be a hexadecimal integer (base sixteen). The hexadecimal digits include a or A through f or F with decimal values ten through fifteen, respectively. For example, the number twelve can be written 12, 014, or 0XC.

3.2.5.2 Character Literals

A character literal is one or more characters enclosed in single quotes, as in ’x’. Character literals have type **char**.

A character is an 8-bit quantity with a numerical value between 0 and 255 (decimal). The value of a space, alphabetic, digit, or graphic character literal is the numerical value of the character as defined in the ISO Latin-1 (8859.1) character set standard (See Table 3-2 on page 3-4, Table 3-3 on page 3-4, and Table 3-4 on page 3-5). The value of a null is 0. The value of a formatting character literal is the numerical value of the character as defined in the ISO 646 standard (see Table 3-5 on page 3-6). The meaning of all other characters is implementation-dependent.

Nongraphic characters must be represented using escape sequences as defined below in Table 3-9. Note that escape sequences must be used to represent single quote and backslash characters in character literals.

Table 3-9 Escape Sequences

Description	Escape Sequence
newline	\n
horizontal tab	\t
vertical tab	\v
backspace	\b
carriage return	\r
form feed	\f
alert	\a
backslash	\\
question mark	\?
single quote	\'
double quote	\"
octal number	\ooo
hexadecimal number	\xhh
unicode character	\uhhhh

If the character following a backslash is not one of those specified, the behavior is undefined. An escape sequence specifies a single character.

The escape \ooo consists of the backslash followed by one, two, or three octal digits that are taken to specify the value of the desired character. The escape \xhh consists of the backslash followed by x followed by one or two hexadecimal digits that are taken to specify the value of the desired character.

The escape \uhhhh consists of a backslash followed by the character ‘u’, followed by one, two, three or four hexadecimal digits. This represents a unicode character literal. Thus the literal “\u002E” represents the unicode period ‘.’ character and the literal “\u3BC” represents the unicode greek small letter ‘mu’. The \u escape is valid only with wchar and wstring types. Since wide string literal is defined as a sequence of wide character literals a sequence of \u literals can be used to define a wide string literal. Attempt to set a char type to a \u defined literal or a string type to a sequence of \u literals result in an error.

A sequence of octal or hexadecimal digits is terminated by the first character that is not an octal digit or a hexadecimal digit, respectively. The value of a character constant is implementation dependent if it exceeds that of the largest char.

Wide character literals have an **L** prefix, for example:

```
const wchar C1 = L'X';
```

Attempts to assign a wide character literal to a non-wide character constant or to assign a non-wide character literal to a wide character constant result in a compile-time diagnostic.

Both wide and non-wide character literals must be specified using characters from the ISO 8859-1 character set.

3.2.5.3 Floating-point Literals

A floating-point literal consists of an integer part, a decimal point, a fraction part, an e or E, and an optionally signed integer exponent. The integer and fraction parts both consist of a sequence of decimal (base ten) digits. Either the integer part or the fraction part (but not both) may be missing; either the decimal point or the letter e (or E) and the exponent (but not both) may be missing.

3.2.5.4 String Literals

A string literal is a sequence of characters (as defined in Section 3.2.5.2, “Character Literals,” on page 3-9) surrounded by double quotes, as in "...".

Adjacent string literals are concatenated. Characters in concatenated strings are kept distinct. For example,

```
"\xA" "B"
```

contains the two characters '\xA' and 'B' after concatenation (and not the single hexadecimal character '\xAB').

The size of a string literal is the number of character literals enclosed by the quotes, after concatenation. The size of the literal is associated with the literal. Within a string, the double quote character **"** must be preceded by a \.

A string literal may not contain the character '\0'.

Wide string literals have an L prefix, for example:

```
const wstring S1 = L"Hello";
```

Attempts to assign a wide string literal to a non-wide string constant or to assign a non-wide string literal to a wide string constant result in a compile-time diagnostic.

Both wide and non-wide string literals must be specified using characters from the ISO 8859-1 character set.

A wide string literal shall not contain the wide character with value zero.

3.2.5.5 Fixed-Point Literals

A fixed-point decimal literal consists of an integer part, a decimal point, a fraction part and a d or D. The integer and fraction parts both consist of a sequence of decimal (base 10) digits. Either the integer part or the fraction part (but not both) may be missing; the decimal point (but not the letter d (or D)) may be missing.

3.3 Preprocessing

OMG IDL preprocessing, which is based on ANSI C++ preprocessing, provides macro substitution, conditional compilation, and source file inclusion. In addition, directives are provided to control line numbering in diagnostics and for symbolic debugging, to generate a diagnostic message with a given token sequence, and to perform implementation-dependent actions (the **#pragma** directive). Certain predefined names are available. These facilities are conceptually handled by a preprocessor, which may or may not actually be implemented as a separate process.

Lines beginning with # (also called “directives”) communicate with this preprocessor. White space may appear before the #. These lines have syntax independent of the rest of OMG IDL; they may appear anywhere and have effects that last (independent of the OMG IDL scoping rules) until the end of the translation unit. The textual location of OMG IDL-specific pragmas may be semantically constrained.

A preprocessing directive (or any line) may be continued on the next line in a source file by placing a backslash character (“\”), immediately before the newline at the end of the line to be continued. The preprocessor effects the continuation by deleting the backslash and the newline before the input sequence is divided into tokens. A backslash character may not be the last character in a source file.

A preprocessing token is an OMG IDL token (see Section 3.2.1, “Tokens,” on page 3-6), a file name as in a **#include** directive, or any single character other than white space that does not match another preprocessing token.

The primary use of the preprocessing facilities is to include definitions from other OMG IDL specifications. Text in files included with a **#include** directive is treated as if it appeared in the including file. A complete description of the preprocessing facilities may be found in *The Annotated C++ Reference Manual*. The #pragma directive that is used to include RepositoryIds is described in Section 10.6, “RepositoryIds,” on page 10-39.

3.4 OMG IDL Grammar

```
(1)   <specification>  ::= <definition>+
(2)    <definition>    ::= <type_dcl> ";"
                         |  <const_dcl> ";"
                         |  <except_dcl> ";"
                         |  <interface> ";"
                         |  <module> ";"
                         |  <value> ";"
(3)      <module>      ::= "module" <identifier> "{" <definition>+ "}"
```

```
(4)                 <interface>  ::= <interface_dcl>
                                 |   <forward_dcl>
(5)               <interface_dcl>  ::= <interface_header> “{” <interface_body> “}”
(6)               <forward_dcl>  ::= [ “abstract” ] “interface” <identifier>
(7)            <interface_header>  ::= [ “abstract” ] “interface” <identifier>
                                       [ <interface_inheritance_spec> ]
(8)              <interface_body>  ::= <export>*
(9)                    <export>  ::= <type_dcl> “;”
                                 |   <const_dcl> “;”
                                 |   <except_dcl> “;”
                                 |   <attr_dcl> “;”
                                 |   <op_dcl> “;”
(10)<interface_inheritance_spec>::=“:” <interface_name>
                                       { “,” <interface_name> }*
(11)            <interface_name>  ::= <scoped_name>
(12)              <scoped_name>  ::= <identifier>
                                 |   “::” <identifier>
                                 |   <scoped_name> “::” <identifier>
(13)                     <value>  ::= ( <value_dcl> | <value_abs_dcl> |
                                       <value_box_dcl> |  <value_forward_dcl>)
(14)         <value_forward_dcl>  ::= [ “abstract” ] “valuetype” <identifier>
(15)             <value_box_dcl>  ::= “valuetype”  <identifier> <type_spec>
(16)             <value_abs_dcl>  ::= “abstract” “valuetype” <identifier>
                                       [ <value_inheritance_spec> ]
                                       “{“ <export>* “}”
(17)                   <value_dcl>  ::= <value_header> “{“  < value_element>* “}”
(18)                <value_header>  ::= [“custom” ] “valuetype” <identifier>
                                       [ <value_inheritance_spec> ]
(19)<value_inheritance_spec>  ::= [ “:” [ “truncatable” ] <value_name>
                                       { “,”  <value_name> }* ]
                                       [ “supports” <interface_name>
                                       { “,” <interface_name> }*  ]
(20)                 <value_name>  ::= <scoped_name>
(21)              <value_element>  ::= <export> | < state_member> | <init_dcl>
(22)               <state_member>  ::= ( “public” | “private” )
                                       <type_spec> <declarators> “;”
(23)                    <init_dcl>  ::= “factory” <identifier>
                                       “(“  [ <init_param_decls> ]  “)” “;”
(24)              <init_param_decls>  ::= <init_param_decl> { “,” <init_param_decl> }
(25)               <init_param_decl>  ::= <init_param_attribute> <param_type_spec>
                                       <simple_declarator>
(26)          <init_param_attribute>  ::= “in”
(27)                  <const_dcl>  ::= <const_dcl>::=“const” <const_type>
                                       <identifier> “=” <const_exp>
(28)                 <const_type>  ::= <integer_type>
                                 |   <char_type>
                                 |   <wide_char_type>
```

```
                                    |   <boolean_type>
                                    |   <floating_pt_type>
                                    |   <string_type>
                                    |   <wide_string_type>
                                    |   <fixed_pt_const_type>
                                    |   <scoped_name>
                                    |   <octet_type>
(29)              <const_exp>       ::= <or_expr>
(30)              <or_expr>         ::= <xor_expr>
                                    |   <or_expr> "|" <xor_expr>
(31)              <xor_expr>        ::= <and_expr>
                                    |   <xor_expr> "^" <and_expr>
(32)              <and_expr>        ::= <shift_expr>
                                    |   <and_expr> "&" <shift_expr>
(33)              <shift_expr>      ::= <add_expr>
                                    |   <shift_expr> ">>" <add_expr>
                                    |   <shift_expr> "<<" <add_expr>
(34)              <add_expr>        ::= <mult_expr>
                                    |   <add_expr> "+" <mult_expr>
                                    |   <add_expr> "-" <mult_expr>
(35)              <mult_expr>       ::= <unary_expr>
                                    |   <mult_expr> "*" <unary_expr>
                                    |   <mult_expr> "/" <unary_expr>
                                    |   <mult_expr> "%" <unary_expr>
(36)              <unary_expr>      ::= <unary_operator> <primary_expr>
                                    |   <primary_expr>
(37)              <unary_operator>  ::= "-"
                                    |   "+"
                                    |   "~"
(38)              <primary_expr>    ::= <scoped_name>
                                    |   <literal>
                                    |   "(" <const_exp> ")"
(39)              <literal>         ::= <integer_literal>
                                    |   <string_literal>
                                    |   <wide_string_literal>
                                    |   <character_literal>
                                    |   <wide_character_literal>
                                    |   <fixed_pt_literal>
                                    |   <floating_pt_literal>
                                    |   <boolean_literal>
(40)              <boolean_literal> ::= "TRUE"
                                    |   "FALSE"
(41)              <positive_int_const> ::= <const_exp>
(42)              <type_dcl>        ::= "typedef" <type_declarator>
                                    |   <struct_type>
                                    |   <union_type>
                                    |   <enum_type>
                                    |   "native" <simple_declarator>
```

```
(43)          <type_declarator>  ::= <type_spec> <declarators>
(44)                <type_spec>  ::= <simple_type_spec>
                                 |   <constr_type_spec>
(45)         <simple_type_spec>  ::= <base_type_spec>
                                 |   <template_type_spec>
                                 |   <scoped_name>
(46)          <base_type_spec>   ::= <floating_pt_type>
                                 |   <integer_type>
                                 |   <char_type>
                                 |   <wide_char_type>
                                 |   <boolean_type>
                                 |   <octet_type>
                                 |   <any_type>
                                 |   <object_type>
                                 |   <value_base_type>
(47)      <template_type_spec>   ::= <sequence_type>
                                 |   <string_type>
                                 |   <wide_string_type>
                                 |   <fixed_pt_type>
(48)          <constr_type_spec> ::= <struct_type>
                                 |   <union_type>
                                 |   <enum_type>
(49)              <declarators>  ::= <declarator> { “,” <declarator> }*
(50)              <declarator>   ::= <simple_declarator>
                                 |   <complex_declarator>
(51)         <simple_declarator> ::= <identifier>
(52)        <complex_declarator> ::= <array_declarator>
(53)          <floating_pt_type> ::= “float”
                                 |   “double”
                                 |   “long” “double”
(54)              <integer_type> ::= <signed_int>
                                 |   <unsigned_int>
(55)                <signed_int> ::= <signed_short_int>
                                 |   <signed_long_int>
                                 |   <signed_longlong_int>
(56)         <signed_short_int>  ::= “short”
(57)          <signed_long_int>  ::= “long”
(58)       <signed_longlong_int> ::= “long” “long”
(59)              <unsigned_int> ::= <unsigned_short_int>
                                 |   <unsigned_long_int>
                                 |   <unsigned_longlong_int>
(60)       <unsigned_short_int>  ::= “unsigned” “short”
(61)        <unsigned_long_int>  ::= “unsigned” “long”
(62)     <unsigned_longlong_int> ::= “unsigned” “long” “long”
(63)              <char_type>    ::= “char”
(64)         <wide_char_type>    ::= “wchar”
(65)              <boolean_type> ::= “boolean”
```

```
(66)          <octet_type>  ::= "octet"
(67)          <any_type>    ::= "any"
(68)          <object_type> ::= "Object"
(69)          <struct_type> ::= "struct" <identifier> "{" <member_list> "}"
(70)          <member_list> ::= <member>+
(71)          <member>      ::= <type_spec> <declarators> ";"
(72)          <union_type>  ::= "union" <identifier> "switch"
                                "(" <switch_type_spec> ")"
                                "{" <switch_body> "}"
(73)   <switch_type_spec>   ::= <integer_type>
                              |  <char_type>
                              |  <boolean_type>
                              |  <enum_type>
                              |  <scoped_name>
(74)          <switch_body> ::= <case>+
(75)          <case>        ::= <case_label>+ <element_spec> ";"
(76)          <case_label>  ::= "case" <const_exp> ":"
                              |  "default" ":"
(77)          <element_spec>::= <type_spec> <declarator>
(78)          <enum_type>   ::= "enum" <identifier>
                                "{" <enumerator> { "," <enumerator> }* "}"
(79)          <enumerator>  ::= <identifier>
(80)       <sequence_type>  ::= "sequence" "<" <simple_type_spec> ","
                                <positive_int_const> ">"
                              |  "sequence" "<" <simple_type_spec> ">"
(81)       <string_type>    ::= "string" "<" <positive_int_const> ">"
                              |  "string"
(82)    <wide_string_type>  ::= "wstring" "<" <positive_int_const> ">"
                              |  "wstring"
(83)     <array_declarator> ::= <identifier> <fixed_array_size>+
(84)     <fixed_array_size> ::= "[" <positive_int_const> "]"
(85)          <attr_dcl>    ::= [ "readonly" ] "attribute"
                                <param_type_spec> <simple_declarator>
                                {  "," <simple_declarator> }*
(86)          <except_dcl>  ::= "exception" <identifier> "{" <member>* "}"
(87)          <op_dcl>      ::= [ <op_attribute> ] <op_type_spec>
                                <identifier> <parameter_dcls>
                                [ <raises_expr> ] [ <context_expr> ]
(88)          <op_attribute>::= "oneway"
(89)          <op_type_spec>::= <param_type_spec>
                              |  "void"
(90)       <parameter_dcls> ::= "(" <param_dcl> { "," <param_dcl> }* ")"
                              |  "(" ")"
(91)          <param_dcl>   ::= <param_attribute> <param_type_spec>
                                <simple_declarator>
(92)       <param_attribute>::= "in"
```

```
                                  |   “out”
                                  |   “inout”
(93)              <raises_expr>   ::= “raises” “(” <scoped_name>
                                      { “,”  <scoped_name> }* “)”
(94)              <context_expr>  ::= “context” “(” <string_literal>
                                      { “,” <string_literal> }* “)”
(95)           <param_type_spec>  ::= <base_type_spec>
                                  |   <string_type>
                                  |   <wide_string_type>
                                  |   <scoped_name>
(96)              <fixed_pt_type> ::= “fixed” “<“ <positive_int_const> “,”
                                      <positive_int_const> “>”
(97)     <fixed_pt_const_type>    ::= “fixed”
(98)          <value_base_type>   ::= “ValueBase”
```

3.5 OMG IDL Specification

An OMG IDL specification consists of one or more type definitions, constant definitions, exception definitions, or module definitions. The syntax is:

```
<specification>   ::= <definition>+

<definition>      ::= <type_dcl> “;”
                  |   <const_dcl> “;”
                  |   <except_dcl> “;”
                  |   <interface> “;”
                  |   <module> “;”
                  |   <value> “;”
```

See Section 3.9, “Constant Declaration,” on page 3-28, Section 3.10, “Type Declaration,” on page 3-31, and Section 3.11, “Exception Declaration,” on page 3-40 respectively for specifications of **<const_dcl>**, **<type_dcl>**, and **<except_dcl>**.

See Section 3.7, “Interface Declaration,” on page 3-18 for the specification of <interface>.

See Section 3.6, “Module Declaration,” on page 3-17 for the specification of <module>.

See Section 3.8, “Value Declaration,” on page 3-23 for the specification of <value>.

3.6 Module Declaration

A module definition satisfies the following syntax:

<module>::=“module” <identifier> “{“ <definition>+ “}”

The module construct is used to scope OMG IDL identifiers; see Section 3.14, “CORBA Module,” on page 3-44 for details.

3.7 Interface Declaration

An interface definition satisfies the following syntax:

```
<interface>          ::= <interface_dcl>
                       |  <forward_dcl>

<interface_dcl>      ::= <interface_header> "{" <interface_body> "}"

<forward_dcl>        ::= [ "abstract" ] "interface" <identifier>

<interface_header>   ::= [ "abstract" ] "interface" <identifier>
                           [ <interface_inheritance_spec> ]

<interface_body>     ::= <export>*

<export>             ::= <type_dcl> ";"
                       |  <const_dcl> ";"
                       |  <except_dcl> ";"
                       |  <attr_dcl> ";"
                       |  <op_dcl> ";"
```

3.7.1 Interface Header

The interface header consists of three elements:

- An optional modifier specifying if the interface is an abstract interface.
- The interface name. The name must be preceded by the keyword **interface**, and consists of an identifier that names the interface.
- An optional inheritance specification. The inheritance specification is described in the next section.

The **<identifier>** that names an interface defines a legal type name. Such a type name may be used anywhere an **<identifier>** is legal in the grammar, subject to semantic constraints as described in the following sections. Since one can only hold references to an object, the meaning of a parameter or structure member which is an interface type is as a *reference* to an object supporting that interface. Each language binding describes how the programmer must represent such interface references.

Abstract interfaces have slightly different rules and semantics from "regular" interfaces as described in Chapter 6, "Abstract Interface Semantics". They also follow different language mapping rules.

3.7.2 Interface Inheritance Specification

The syntax for inheritance is as follows:

```
<interface_inheritance_spec>::= ":" <interface_name>
                                    {"," <interface_name>}*

<interface_name>            ::= <scoped_name>
```

<scoped_name> **::= <identifier>**
| "::" <identifier>
| <scoped_name> "::" <identifier>

Each **<scoped_name>** in an **<interface_inheritance_spec>** must denote a previously defined interface. See Section 3.7.5, "Interface Inheritance," on page 3-20 for the description of inheritance.

3.7.3 Interface Body

The interface body contains the following kinds of declarations:

- Constant declarations, which specify the constants that the interface exports; constant declaration syntax is described in Section 3.9, "Constant Declaration," on page 3-28.
- Type declarations, which specify the type definitions that the interface exports; type declaration syntax is described in Section 3.10, "Type Declaration," on page 3-31.
- Exception declarations, which specify the exception structures that the interface exports; exception declaration syntax is described in Section 3.11, "Exception Declaration," on page 3-40.
- Attribute declarations, which specify the associated attributes exported by the interface; attribute declaration syntax is described in Section 3.13, "Attribute Declaration," on page 3-43.
- Operation declarations, which specify the operations that the interface exports and the format of each, including operation name, the type of data returned, the types of all parameters of an operation, legal exceptions which may be returned as a result of an invocation, and contextual information which may affect method dispatch; operation declaration syntax is described in Section 3.12, "Operation Declaration," on page 3-41.

Empty interfaces are permitted (that is, those containing no declarations).

Some implementations may require interface-specific pragmas to precede the interface body.

3.7.4 Forward Declaration

A forward declaration declares the name of an interface without defining it. This permits the definition of interfaces that refer to each other. The syntax consists simply of the keyword **interface** followed by an **<identifier>** that names the interface. The actual definition must follow later in the specification.

Multiple forward declarations of the same interface name are legal.

It is illegal to inherit from a forward-declared interface whose definition has not yet been seen:

```
module Example {
    interface base;              // Forward declaration

    // ...

    interface derived : base {};// Error
    interface base {};           // Define base
    interface derived : base {};// OK
};
```

3.7.5 Interface Inheritance

An interface can be derived from another interface, which is then called a *base* interface of the derived interface. A derived interface, like all interfaces, may declare new elements (constants, types, attributes, exceptions, and operations). In addition, unless redefined in the derived interface, the elements of a base interface can be referred to as if they were elements of the derived interface. The name resolution operator (“::”) may be used to refer to a base element explicitly; this permits reference to a name that has been redefined in the derived interface.

A derived interface may redefine any of the type, constant, and exception names which have been inherited; the scope rules for such names are described in Section 3.14, “CORBA Module,” on page 3-44.

An interface is called a direct base if it is mentioned in the **<interface_inheritance_spec>** and an indirect base if it is not a direct base but is a base interface of one of the interfaces mentioned in the **<interface_inheritance_spec>**.

An interface may be derived from any number of base interfaces. Such use of more than one direct base interface is often called multiple inheritance. The order of derivation is not significant.

An abstract interface may only inherit from other abstract interfaces.

An interface may not be specified as a direct base interface of a derived interface more than once; it may be an indirect base interface more than once. Consider the following example:

```
interface A { ... }
interface B: A { ... }
interface C: A { ... }
interface D: B, C { ... }
interface E: A, B { ... };                    // OK
```

The relationships between these interfaces is shown in Figure 3-1. This “diamond” shape is legal, as is the definition of E on the right.

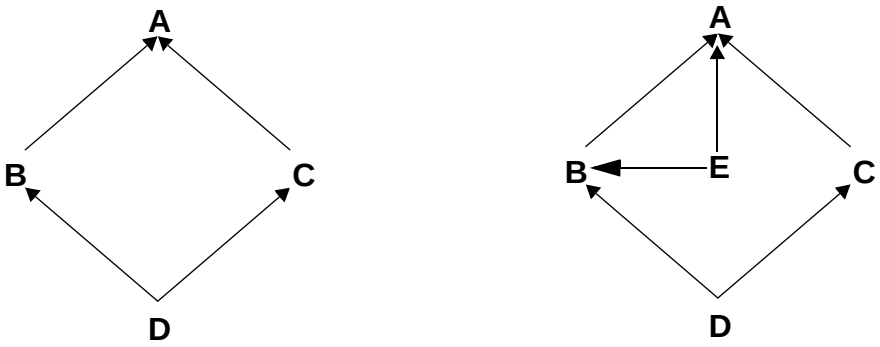

Figure 3-1 Legal Multiple Inheritance Example

References to base interface elements must be unambiguous. A Reference to a base interface element is ambiguous if the name is declared as a constant, type, or exception in more than one base interface. Ambiguities can be resolved by qualifying a name with its interface name (that is, using a **<scoped_name>**). It is illegal to inherit from two interfaces with the same operation or attribute name, or to redefine an operation or attribute name in the derived interface.

So for example in:

```
interface A {
    typedef long L1;
    short opA(in L1 l_1);
};

interface B {
    typedef short L1;
    L1 opB(in long l);
};

interface C: B, A {
    typedef L1 L2;          // Error: L1 ambiguous
    typedef A::L1 L3;       // A::L1 is OK
    B::L1 opC(in L3 l_3);   // all OK no ambiguities
};
```

References to constants, types, and exceptions are bound to an interface when it is defined (i.e., replaced with the equivalent global **<scoped_name>**s). This guarantees that the syntax and semantics of an interface are not changed when the interface is a base interface for a derived interface. Consider the following example:

```
const long L = 3;

interface A {
    typedef float coord[L]:
    void f (in coord s);          // s has three floats
};

interface B {
    const long L = 4;
};

interface C: B, A { };                    // what is C::f()'s signature?
```

The early binding of constants, types, and exceptions at interface definition guarantees that the signature of operation **f** in interface **C** is

```
typedef float coord[3];
void f (in coord s);
```

which is identical to that in interface **A**. This rule also prevents redefinition of a constant, type, or exception in the derived interface from affecting the operations and attributes inherited from a base interface.

Interface inheritance causes all identifiers in the closure of the inheritance tree to be imported into the current naming scope. A type name, constant name, enumeration value name, or exception name from an enclosing scope can be redefined in the current scope. An attempt to use an ambiguous name without qualification produces a compilation error. Thus in

```
interface A {
    typedef string<128> string_t;
};

interface B {
    typedef string<256> string_t;
};

interface C: A, B {
    attribute string_t      Title;      // Error: string_t ambiguous
    attribute A::string_t   Name;       // OK
    attribute B::string_t   City;       // OK
};
```

Operation and attribute names are used at run-time by both the stub and dynamic interfaces. As a result, all operations attributes that might apply to a particular object must have unique names. This requirement prohibits redefining an operation or attribute name in a derived interface, as well as inheriting two operations or attributes with the same name.

```
interface A {
    void make_it_so();
};

interface B: A {
    short make_it_so(in long times);   // Error: redefinition of make_it_so
};
```

3.8 Value Declaration

There are several kinds of value type declarations: “regular” value types, boxed value types, abstract value types, and forward declarations.

A value declaration satisfies the following syntax:

```
<value>     ::= ( <value_dcl>
            |   <value_abs_dcl>
            |   <value_box_dcl>
            |   <value_forward_dcl>) “;”
```

3.8.1 Regular Value Type

A regular value type satisfies the following syntax:

```
<value_dcl>     ::= <value_header> “{“  < value_element >* “}”

<value_header>  ::= [“custom” ] “valuetype” <identifier>
                    [ <value_inheritance_spec> ]

<value_element> ::= <export>
                |   < state_member>
                |   <init_dcl>
```

3.8.1.1 Value Header

The value header consists of two elements:

- The value type’s name and optional modifier specifying whether the value type uses custom marshaling.
- An optional value inheritance specification. The value inheritance specification is described in the next section.

3.8.1.2 Value Element

A value can contain all the elements that an interface can as well as the definition of state members, and initializers for that state.

3.8.1.3 Value Inheritance Specification

```
<value_inheritance_spec> ::=  [ “:” [ “truncatable” ] <value_name>
                                { “,” <value_name> }* ]
```

[“supports” <interface_name>
{ “,” interface_name> }*]

<value_name> ::= <scoped_name>

Each **<value_name>** and **<interface_name>** in a **<value_inheritance_spec>** must denote previously defined value type or interface. See Section 3.8.5, “Valuetype Inheritance,” on page 3-27 for the description of value type inheritance.

The **truncatable** modifier may not be used if the value type being defined is a custom value.

3.8.1.4 State Members

<state_member> ::=(“public” | “private”) <type_spec> <declarators> “;”

Each <state_member> defines an element of the state which is marshaled and sent to the receiver when the value type is passed as a parameter. A state member is either public or private. The annotation directs the language mapping to hide or expose the different parts of the state to the clients of the value type. The private part of the state is only accessible to the implementation code and the marshaling routines.

Note that certain programming languages may not have the built in facilities needed to distinguish between the public and private members. In these cases, the language mapping specifies the rules that programmers are responsible for following.

3.8.1.5 Initializers

<init_dcl> ::= “factory” <identifier>
“(“ [<init_param_decls>] “)” “;”

<init_param_decls> ::= <init_param_decl> { “,” <init_param_decl> }

<init_param_decl> ::= <init_param_attribute> <param_type_spec>
<simple_declarator>

<init_param_attribute> ::= “in”

In order to ensure portability of value implementations, designers may also define the signatures of initializers (or constructors) for non abstract value types. Syntactically these look like local operation signatures except that they are prefixed with the keyword **factory**, have no return type, and must use only in parameters. There may be any number of factory declarations. The names of the initializers are part of the name scope of the value type.

If no initializers are specified in IDL, the value type does not provide a portable way of creating a runtime instance of its type. There is no default initializer. This allows the definition of IDL value types which are not intended to be directly instantiated by client code.

3.8.1.6 *Value Type Example*

```
interface Tree {
    void print()
};

valuetype WeightedBinaryTree {
    // state definition
        private unsigned long weight;
        private WeightedBinaryTree left;
        private WeightedBinaryTree right;
    // initializer
        factory init(in unsigned long w);
    // local operations
        WeightSeq pre_order();
        WeightSeq post_order();
};
valuetype WTree: WeightedBinaryTree supports Tree {};
```

3.8.2 *Boxed Value Type*

<value_box_dcl> ::= “valuetype” <identifier> <type_spec>

It is often convenient to define a value type with no inheritance or operations and with a single state member. A shorthand IDL notation is used to simplify the use of value types for this kind of simple containment, referred to as a “value box”.

This is particularly useful for strings and sequences. Basically one does not have to create what is in effect an additional namespace that will contain only one name.

An example is the following IDL:

```
module Example {
    interface Foo {
        ... /* anything */
    };
    valuetype FooSeq sequence<Foo>;
    interface Bar {
        void doIt (in FooSeq seq1);
    };
};
```

The above IDL provides similar functionality to writing the following IDL. However the type identities (repository ID’s) would be different.

```
module Example {
    interface Foo {
        ... /* anything */
    };
    valuetype FooSeq {
        public sequence<Foo> data;
    };
    interface Bar {
        void doIt (in FooSeq seq);
    };
};
```

The former is easier to manipulate after it is mapped to a concrete programming language.

The declaration of a boxed value type does not open a new scope.Thus a construction such as:

```
valuetype FooSeq sequence <FooSeq>;
```

is not legal IDL. The identifier being declared as a boxed value type cannot be used subsequent to its initial use and prior to the completion of the boxed value declaration.

3.8.3 Abstract Value Type

<value_abs_dcl> ::= “abstract” “valuetype” <identifier>
[<value_inheritance_spec>] “{“ <export>* “}”

Value types may also be abstract. They are called abstract because an abstract value type may not be instantiated. No <state_member> or <initializers> may be specified. However, local operations may be specified. Essentially they are a bundle of operation signatures with a purely local implementation.

Note that a concrete value type with an empty state is not an abstract value type.

3.8.4 Value Forward Declaration

<value_forward_dcl> ::= [“abstract”] “valuetype” <identifier>

A forward declaration declares the name of a value type without defining it. This permits the definition of value types that refer to each other. The syntax consists simply of the keyword **valuetype** followed by an **<identifier>** that names the value type. The actual definition must follow later in the specification.

Multiple forward declarations of the same value type name are legal.

Boxed value types cannot be forward declared; such a forward declaration would refer to a normal value type.

It is illegal to inherit from a forward-declared value type whose definition has not yet been seen.

3.8.5 Valuetype Inheritance

The terminology that is used to describe value type inheritance is directly analogous to that used to describe interface inheritance (see Section 3.7.5, “Interface Inheritance,” on page 3-20).

The name scoping and name collision rules for valuetypes are identical to those for interfaces. In addition, no valuetype may be specified as a direct abstract base of a derived valuetype more than once; it may be an indirect abstract base more than once. See Section 3.7.5, “Interface Inheritance,” on page 3-20 for a detailed description of the analogous properties for interfaces.

Values may be derived from other values and can support an interface and any number of abstract interfaces.

Once implementation (state) is specified at a particular point in the inheritance hierarchy, all derived value types (which must of course implement the state) may only derive from a single (concrete) value type. They can however derive from other additional abstract values and support an additional interface.

The single immediate base concrete value type, if present, must be the first element specified in the inheritance list of the value declaration’s IDL. It may be followed by other abstract values from which it inherits. The interface and abstract interfaces that it supports are listed following the **supports** keyword.

A stateful value that derives from another stateful value may specify that it is truncatable. This means that it is to “truncate” (see Section 5.2.5.2, “Value instance -> Value type,” on page 5-5) an instance to be an instance of any of its truncatable parent (stateful) value types under certain conditions. Note that all the intervening types in the inheritance hierarchy must be truncatable in order for truncation to a particular type to be allowed.

Because custom values require an exact type match between the sending and receiving context, **truncatable** may not be specified for a custom value type.

Non-custom value types may not (transitively) inherit from custom value types.

Boxed value types may not be derived from, nor may they derive from anything else.

These rules are summarized in the following table:

Table 3-10 Allowable Inheritance Relationships

May inherit from:	Interface	Abstract Interface	Abstract Value	Stateful Value	Boxed value
Interface	multiple	multiple	no	no	no
Abstract Interface	no	multiple	no	no	no
Abstract Value	supports	supports	multiple	no	no
Stateful Value	supports single	supports	multiple	single (may be truncatable)	no
Boxed Value	no	no	no	no	no

3.9 Constant Declaration

This section describes the syntax for constant declarations.

3.9.1 Syntax

The syntax for a constant declaration is:

```
<const_dcl>        ::= "const" <const_type> <identifier>
                       "=" <const_exp>

<const_type>       ::= <integer_type>
                   |   <char_type>
                   |   <wide_char_type>
                   |   <boolean_type>
                   |   <floating_pt_type>
                   |   <string_type>
                   |   <wide_string_type>
                   |   <fixed_pt_const_type>
                   |   <scoped_name>
                   |   <octet_type>

<const_exp>        ::= <or_expr>

<or_expr>          ::= <xor_expr>
                   |   <or_expr> "|" <xor_expr>

<xor_expr>         ::= <and_expr>
                   |   <xor_expr> "^" <and_expr>

<and_expr>         ::= <shift_expr>
                   |   <and_expr> "&" <shift_expr>

<shift_expr>       ::= <add_expr>
                   |   <shift_expr> ">>" <add_expr>
                   |   <shift_expr> "<<" <add_expr>

<add_expr>         ::= <mult_expr>
                   |   <add_expr> "+" <mult_expr>
                   |   <add_expr> "-" <mult_expr>

<mult_expr>        ::= <unary_expr>
                   |   <mult_expr> "*" <unary_expr>
                   |   <mult_expr> "/" <unary_expr>
                   |   <mult_expr> "%" <unary_expr>

<unary_expr>       ::= <unary_operator> <primary_expr>
                   |   <primary_expr>

<unary_operator>   ::= "-"
                   |   "+"
                   |   "~"

<primary_expr>     ::= <scoped_name>
                   |   <literal>
                   |   "(" <const_exp> ")"
```

```
<literal>            ::= <integer_literal>
                     |   <string_literal>
                     |   <character_literal>
                     |   <floating_pt_literal>
                     |   <boolean_literal>
<boolean_literal>    ::= "TRUE"
                     |   "FALSE"
<positive_int_const> ::= <const_exp>
```

3.9.2 Semantics

The **<scoped_name>** in the **<const_type>** production must be a previously defined name of an **<integer_type>**, **<char_type>**, **<wide_char_type>**, **<boolean_type>**, **<floating_pt_type>**, **<string_type>, <wide_string_type>**, **<octet_type>**, or **<enum_type>** constant.

An infix operator can combine two integers, floats or fixeds, but not mixtures of these. Infix operators are applicable only to integer, float and fixed types.

If the type of an integer constant is **long** or **unsigned long**, then each subexpression of the associated constant expression is treated as an **unsigned long** by default, or a signed **long** for negated literals or negative integer constants. It is an error if any subexpression values exceed the precision of the assigned type (**long** or **unsigned long**), or if a final expression value (of type **unsigned long**) exceeds the precision of the target type (**long**).

If the type of an integer constant is **long long** or **unsigned long long**, then each subexpression of the associated constant expression is treated as an **unsigned long long** by default, or a signed **long long** for negated literals or negative integer constants. It is an error if any subexpression values exceed the precision of the assigned type (**long long** or **unsigned long long**), or if a final expression value (of type **unsigned long long**) exceeds the precision of the target type (**long long**).

If the type of a floating-point constant is **double**, then each subexpression of the associated constant expression is treated as a **double.** It is an error if any subexpression value exceeds the precision of **double**.

If the type of a floating-point constant is **long double**, then each subexpression of the associated constant expression is treated as a **long double**. It is an error if any subexpression value exceeds the precision of **long double**.

Fixed-point decimal constant expressions are evaluated as follows. A fixed-point literal has the apparent number of total and fractional digits, except that leading and trailing zeros are factored out, including non-significant zeros before the decimal point. For example, **0123.450d** is considered to be **fixed<5,2>** and **3000.00D** is **fixed<1,-3>**. Prefix operators do not affect the precision; a prefix **+** is optional, and does not change

the result. The upper bounds on the number of digits and scale of the result of an infix expression, **fixed<d1,s1> op fixed<d2,s2>**, are shown in the following table:

Op	Result: fixed<d,s>
+	fixed<max(d1-s1,d2-s2) + max(s1,s2) + 1, max(s1,s2)>
-	fixed<max(d1-s1,d2-s2) + max(s1,s2) + 1, max(s1,s2)>
*	fixed<d1+d2, s1+s2>
/	**fixed<(d1-s1+s2) + s_{inf}, s_{inf}>**

A quotient may have an arbitrary number of decimal places, denoted by a scale of s_{inf}. The computation proceeds pairwise, with the usual rules for left-to-right association, operator precedence, and parentheses. All intermediate computations shall be performed using double precision (i.e. 62 digit) arithmetic. If an individual computation between a pair of fixed-point literals actually generates more than 31 significant digits, then a 31-digit result is retained as follows:

fixed<d,s> => fixed<31, 31-d+s>

Leading and trailing zeros are not considered significant. The omitted digits are discarded; rounding is not performed. The result of the individual computation then proceeds as one literal operand of the next pair of fixed-point literals to be computed.

Unary (**+ -**) and binary (*** / + -**) operators are applicable in floating-point and fixed-point expressions. Unary (**+ - ~**) and binary (*** / % + - << >> & | ^**) operators are applicable in integer expressions.

The "~" unary operator indicates that the bit-complement of the expression to which it is applied should be generated. For the purposes of such expressions, the values are 2's complement numbers. As such, the complement can be generated as follows:

Integer Constant Expression Type	Generated 2's Complement Numbers
long	long -(value+1)
unsigned long	unsigned long (2**32-1) - value
long long	long long -(value+1)
unsigned long long	unsigned long (2**64-1) - value

The "%" binary operator yields the remainder from the division of the first expression by the second. If the second operand of "%" is 0, the result is undefined; otherwise

(a/b)*b + a%b

is equal to a. If both operands are nonnegative, then the remainder is nonnegative; if not, the sign of the remainder is implementation dependent.

The “<<”binary operator indicates that the value of the left operand should be shifted left the number of bits specified by the right operand, with 0 fill for the vacated bits. The right operand must be in the range 0 <= right operand < 64.

The “>>” binary operator indicates that the value of the left operand should be shifted right the number of bits specified by the right operand, with 0 fill for the vacated bits. The right operand must be in the range 0 <= right operand < 64.

The “&” binary operator indicates that the logical, bitwise AND of the left and right operands should be generated.

The “|” binary operator indicates that the logical, bitwise OR of the left and right operands should be generated.

The “^” binary operator indicates that the logical, bitwise EXCLUSIVE-OR of the left and right operands should be generated.

<positive_int_const> must evaluate to a positive integer constant.

An octet constant can be defined using an integer literal or an integer constant expression, for example:

```
const octet O1 = 0x1;
const long L = 3;
const octet O2 = 5 + L;
```

Values for an octet constant outside the range 0 - 255 shall cause a compile-time error.

An enum constant can only be defined using a scoped name for the enumerator. The scoped name is resolved using the normal scope resolution rules Section 3.15, “Names and Scoping,” on page 3-45. For example:

```
enum Color { red, green, blue };
const Color FAVORITE_COLOR = red;

module M {
    enum Size { small, medium, large };
};
const M::Size MYSIZE = M::medium;
```

The constant name for the RHS of an enumerated constant definition must denote one of the enumerators defined for the enumerated type of the constant. For example:

```
const Color col = red;    // is OK but
const Color another = M::medium; // is an error
```

3.10 Type Declaration

OMG IDL provides constructs for naming data types; that is, it provides C language-like declarations that associate an identifier with a type. OMG IDL uses the **typedef** keyword to associate a name with a data type; a name is also associated with a data type via the **struct**, **union**, **enum**, and **native** declarations; the syntax is:

```
<type_dcl>          ::= "typedef" <type_declarator>
                    |   <struct_type>
                    |   <union_type>
                    |   <enum_type>
                    |   "native" <simple_declarator>

<type_declarator>   ::= <type_spec> <declarators>
```

For type declarations, OMG IDL defines a set of type specifiers to represent typed values. The syntax is as follows:

```
<type_spec>         ::= <simple_type_spec>
                    |   <constr_type_spec>

<simple_type_spec>  ::= <base_type_spec>
                    |   <template_type_spec>
                    |   <scoped_name>

<base_type_spec>    ::= <floating_pt_type>
                    |   <integer_type>
                    |   <char_type>
                    |   <wide_char_type>
                    |   <boolean_type>
                    |   <octet_type>
                    |   <any_type>
                    |   <object-type>
                    |   <value_base_type>

<template_type_spec>::= <sequence_type>
                    |   <string_type>
                    |   <wide_string_type>
                    |   <fixed_pt_type>

<constr_type_spec>  ::= <struct_type>
                    |   <union_type>
                    |   <enum_type>

<declarators>       ::= <declarator> { "," <declarator> }*

<declarator>        ::= <simple_declarator>
                    |   <complex_declarator>

<simple_declarator> ::= <identifier>

<complex_declarator>::= <array_declarator>
```

The **<scoped_name>** in **<simple_type_spec>** must be a previously defined type.

As seen above, OMG IDL type specifiers consist of scalar data types and type constructors. OMG IDL type specifiers can be used in operation declarations to assign data types to operation parameters. The next sections describe basic and constructed type specifiers.

3.10.1 Basic Types

The syntax for the supported basic types is as follows:

```
<floating_pt_type>      ::= "float"
                            | "double"
                            | "long" "double"
<integer_type>:         :=  <signed_int>
                            | <unsigned_int>
<signed_int>            ::= <signed_long_int
                            | <signed_short_int>
                            | <signed_longlong_int>
<signed_long_int>       ::= "long"
<signed_short_int>      ::= "short"
<signed_longlong_int>::= "long" "long"
<unsigned_int>          ::= <unsigned_long_int>
                            | <unsigned_short_int>
                            | <unsigned_longlong_int>
<unsigned_long_int>     ::= "unsigned" "long"
<unsigned_short_int>    ::= "unsigned" "short"
<unsigned_longlong_int>::= "unsigned" "long" "long"
<char_type>             ::= "char"
<wide_char_type>        ::= "wchar"
<boolean_type>          ::= "boolean"
<octet_type>            ::= "octet"
<any_type>              ::= "any"
```

Each OMG IDL data type is mapped to a native data type via the appropriate language mapping. Conversion errors between OMG IDL data types and the native types to which they are mapped can occur during the performance of an operation invocation. The invocation mechanism (client stub, dynamic invocation engine, and skeletons) may signal an exception condition to the client if an attempt is made to convert an illegal value. The standard exceptions which are to be signalled in such situations are defined in Section 3.17, “Standard Exceptions,” on page 3-51.

3.10.1.1 Integer Types

OMG IDL integer types are **short**, **unsigned short**, **long**, **unsigned long**, **long long** and **unsigned long long**, representing integer values in the range indicated below in Table 3-11.

Table 3-11 Range of integer types

short	-2^{15} .. $2^{15} - 1$
long	-2^{31} .. $2^{31} - 1$
long long	-2^{63} .. $2^{63} - 1$
unsigned short	0 .. $2^{16} - 1$

Table 3-11 Range of integer types

unsigned long	0 .. 2^{32} - 1
unsigned long long	0 .. 2^{64} - 1

3.10.1.2 Floating-Point Types

OMG IDL floating-point types are **float**, **double** and **long double**. The **float** type represents IEEE single-precision floating point numbers; the **double** type represents IEEE double-precision floating point numbers.The **long double** data type represents an IEEE double-extended floating-point number, which has an exponent of at least 15 bits in length and a signed fraction of at least 64 bits. See *IEEE Standard for Binary Floating-Point Arithmetic*, ANSI/IEEE Standard 754-1985, for a detailed specification.

3.10.1.3 Char Type

OMG IDL defines a **char** data type that is an 8-bit quantity which (1) encodes a single-byte character from any byte-oriented code set, or (2) when used in an array, encodes a multi-byte character from a multi-byte code set. In other words, an implementation is free to use any code set internally for encoding character data, though conversion to another form may be required for transmission.

The ISO 8859-1 (Latin1) character set standard defines the meaning and representation of all possible graphic characters used in OMG IDL (i.e., the space, alphabetic, digit and graphic characters defined in Table 3-2 on page 3-4, Table 3-3 on page 3-4, and Table 3-4 on page 3-5). The meaning and representation of the null and formatting characters (see Table 3-5 on page 3-6) is the numerical value of the character as defined in the ASCII (ISO 646) standard. The meaning of all other characters is implementation-dependent.

During transmission, characters may be converted to other appropriate forms as required by a particular language binding. Such conversions may change the representation of a character but maintain the character's meaning. For example, a character may be converted to and from the appropriate representation in international character sets.

3.10.1.4 Wide Char Type

OMG IDL defines a **wchar** data type which encodes wide characters from any character set. As with character data, an implementation is free to use any code set internally for encoding wide characters, though, again, conversion to another form may be required for transmission. The size of **wchar** is implementation-dependent.

3.10.1.5 Boolean Type

The **boolean** data type is used to denote a data item that can only take one of the values TRUE and FALSE.

3.10.1.6 Octet Type

The **octet** type is an 8-bit quantity that is guaranteed not to undergo any conversion when transmitted by the communication system.

3.10.1.7 Any Type

The **any** type permits the specification of values that can express any OMG IDL type.

An **any** logically contains a TypeCode (see Section 3.10, “Type Declaration,” on page 3-31) and a value that is described by the TypeCode. Each IDL language mapping provides operations that allow programers to insert and access the TypeCode and value contained in an any.

3.10.2 Constructed Types

The constructed types are:

```
<constr_type_spec>  ::= <struct_type>
                      | <union_type>
                      | <enum_type>
```

Although the IDL syntax allows the generation of recursive constructed type specifications, the only recursion permitted for constructed types is through the use of the **sequence** template type. For example, the following is legal:

```
struct foo {
    long value;
    sequence<foo> chain;
}
```

See Section 3.10.3.1, “Sequences,” on page 3-37 for details of the **sequence** template type.

3.10.2.1 Structures

The structure syntax is:

```
<struct_type>  ::= “struct” <identifier> “{” <member_list> “}”
<member_list>  ::= <member>+
<member>       ::= <type_spec> <declarators> “;”
```

The **<identifier>** in **<struct_type>** defines a new legal type. Structure types may also be named using a **typedef** declaration.

Name scoping rules require that the member declarators in a particular structure be unique. The value of a **struct** is the value of all of its members.

3.10.2.2 Discriminated Unions

The discriminated **union** syntax is:

<union_type> **::=** **“union” <identifier> “switch”**
“(” <switch_type_spec> “)”
“{” <switch_body> “}”

<switch_type_spec> **::=** **<integer_type>**
| <char_type>
| <boolean_type>
| <enum_type>
| <scoped_name>

<switch_body> **::=** **<case>**$^{+}$

<case> **::=** **<case_label>**$^{+}$ **<element_spec> “;”**

<case_label> **::=** **“case” <const_exp> “:”**
| “default” “:”

<element_spec> **::=** **<type_spec> <declarator>**

OMG IDL unions are a cross between the C **`union`** and **`switch`** statements. IDL unions must be discriminated; that is, the union header must specify a typed tag field that determines which union member to use for the current instance of a call. The **<identifier>** following the **union** keyword defines a new legal type. Union types may also be named using a **typedef** declaration. The **<const_exp>** in a **<case_label>** must be consistent with the **<switch_type_spec>**. A **default** case can appear at most once. The **<scoped_name>** in the **<switch_type_spec>** production must be a previously defined **integer**, **char**, **boolean** or **enum** type.

Case labels must match or be automatically castable to the defined type of the discriminator. The complete set of matching rules are shown in Table 3-12.

Table 3-12 Case Label Matching

Discriminator Type	Matched By
long	any integer value in the value range of long
long long	any integer value in the range of long long
short	any integer value in the value range of short
unsigned long	any integer value in the value range of unsigned long
unsigned long long	any integer value in the range of unsigned long long
unsigned short	any integer value in the value range of unsigned short
char	char
wchar	wchar
boolean	TRUE or FALSE
enum	any enumerator for the discriminator enum type

Name scoping rules require that the element declarators in a particular union be unique. If the **<switch_type_spec>** is an **<enum_type>**, the identifier for the enumeration is in the scope of the union; as a result, it must be distinct from the element declarators.

It is not required that all possible values of the union discriminator be listed in the **<switch_body>**. The value of a union is the value of the discriminator together with one of the following:

- If the discriminator value was explicitly listed in a **case** statement, the value of the element associated with that **case** statement;
- If a default **case** label was specified, the value of the element associated with the default **case** label;
- No additional value.

Access to the discriminator and the related element is language-mapping dependent.

3.10.2.3 Enumerations

Enumerated types consist of ordered lists of identifiers. The syntax is:

<enum_type> ::= "enum" <identifier> "{" <enumerator> { "," <enumerator> }* "}"

<enumerator> ::= <identifier>

A maximum of 2^{32} identifiers may be specified in an enumeration; as such, the enumerated names must be mapped to a native data type capable of representing a maximally-sized enumeration. The order in which the identifiers are named in the specification of an enumeration defines the relative order of the identifiers. Any language mapping which permits two enumerators to be compared or defines successor/predecessor functions on enumerators must conform to this ordering relation. The **<identifier>** following the **enum** keyword defines a new legal type. Enumerated types may also be named using a **typedef** declaration.

3.10.3 Template Types

The template types are:

<template_type_spec>::= <sequence_type>
| <string_type>
| <wide_string_type>
| <fixed_pt_type>

3.10.3.1 Sequences

OMG IDL defines the sequence type **sequence**. A sequence is a one-dimensional array with two characteristics: a maximum size (which is fixed at compile time) and a length (which is determined at run time).

The syntax is:

<sequence_type> ::= “sequence” “<” <simple_type_spec> “,” <positive_int_const> “>” | “sequence” “<” <simple_type_spec> “>”

The second parameter in a sequence declaration indicates the maximum size of the sequence. If a positive integer constant is specified for the maximum size, the sequence is termed a bounded sequence. Prior to passing a bounded sequence as a function argument (or as a field in a structure or union), the length of the sequence must be set in a language-mapping dependent manner. After receiving a sequence result from an operation invocation, the length of the returned sequence will have been set; this value may be obtained in a language-mapping dependent manner.

If no maximum size is specified, size of the sequence is unspecified (unbounded). Prior to passing such a sequence as a function argument (or as a field in a structure or union), the length of the sequence, the maximum size of the sequence, and the address of a buffer to hold the sequence must be set in a language-mapping dependent manner. After receiving such a sequence result from an operation invocation, the length of the returned sequence will have been set; this value may be obtained in a language-mapping dependent manner.

A sequence type may be used as the type parameter for another sequence type. For example, the following:

typedef sequence< sequence<long> > Fred;

declares Fred to be of type “unbounded sequence of unbounded sequence of long”. Note that for nested sequence declarations, white space must be used to separate the two “>” tokens ending the declaration so they are not parsed as a single “>>” token.

3.10.3.2 Strings

OMG IDL defines the string type **string** consisting of all possible 8-bit quantities except null. A string is similar to a sequence of char. As with sequences of any type, prior to passing a string as a function argument (or as a field in a structure or union), the length of the string must be set in a language-mapping dependent manner. The syntax is:

<string_type> ::= “string” “<” <positive_int_const> “>” | “string”

The argument to the string declaration is the maximum size of the string. If a positive integer maximum size is specified, the string is termed a bounded string; if no maximum size is specified, the string is termed an unbounded string.

Strings are singled out as a separate type because many languages have special built-in functions or standard library functions for string manipulation. A separate string type may permit substantial optimization in the handling of strings compared to what can be done with sequences of general types.

3.10.3.3 Wstrings

The **wstring** data type represents a sequence of wchar, except the wide character null. The type wstring is similar to that of type string, except that its element type is wchar instead of char. The actual length of a wstring is set at run-time and, if the bounded form is used, must be less than or equal to the bound.

The syntax for defining a wstring is:

<wide_string_type> ::= "wstring" "<" <positive_int_const> ">"
| "wstring"

3.10.3.4 Fixed Type

The **fixed** data type represents a fixed-point decimal number of up to 31 significant digits. The scale factor is a non-negative integer less than or equal to the total number of digits (note that constants with effectively negative scale, such as 10000, are always permitted).

The **fixed** data type will be mapped to the native fixed point capability of a programming language, if available. If there is not a native fixed point type, then the IDL mapping for that language will provide a fixed point data type. Applications that use the IDL fixed point type across multiple programming languages must take into account differences between the languages in handling rounding, overflow, and arithmetic precision.

3.10.4 Complex Declarator

3.10.4.1 Arrays

OMG IDL defines multidimensional, fixed-size arrays. An array includes explicit sizes for each dimension.

The syntax for arrays is:

<array_declarator> ::=<identifier> <fixed_array_size>$^{+}$

<fixed_array_size> ::="[" <positive_int_const> "]"

The array size (in each dimension) is fixed at compile time. When an array is passed as a parameter in an operation invocation, all elements of the array are transmitted.

The implementation of array indices is language mapping specific; passing an array index as a parameter may yield incorrect results.

3.10.5 Native Types

OMG IDL provides a declaration for use by object adapters to define an opaque type whose representation is specified by the language mapping for that object adapter.

The syntax is:

<type_dcl> **::= “native” <simple_declarator>**

<simple_declarator> **::= <identifier>**

This declaration defines a new type with the specified name. A native type is similar to an IDL basic type. The possible values of a native type are language-mapping dependent, as are the means for constructing them and manipulating them. Any interface that defines a native type requires each language mapping to define how the native type is mapped into that programming language.

A native type may be used to define operation parameters and results. However, there is no requirement that values of the type be permitted in remote invocations, either directly or as a component of a constructed type. Any attempt to transmit a value of a native type in a remote invocation may raise the MARSHAL standard exception.

It is recommended that native types be mapped to equivalent type names in each programming language, subject to the normal mapping rules for type names in that language. For example, in a hypothetical Object Adapter IDL module

```
module HypotheticalObjectAdapter {
    native Servant;
    interface HOA {
        Object activate_object(in Servant x);
    };
};
```

the IDL type Servant would map to **HypotheticalObjectAdapter::Servant** in C++ and the **activate_object** operation would map to the following C++ member function signature:

```
CORBA::Object_ptr activate_object(
HypotheticalObjectAdapter::Servant x);
```

The definition of the C++ type **`HypotheticalObjectAdapter::Servant`** would be provided as part of the C++ mapping for the HypotheticalObjectAdapter module.

Note – The native type declaration is provided specifically for use in object adapter interfaces, which require parameters whose values are concrete representations of object implementation instances. It is strongly recommended that it not be used in service or application interfaces. The native type declaration allows object adapters to define new primitive types without requiring changes to the OMG IDL language or to OMG IDL compiler.

3.11 Exception Declaration

Exception declarations permit the declaration of struct-like data structures which may be returned to indicate that an exceptional condition has occurred during the performance of a request. The syntax is as follows:

<except_dcl> **::= “exception” <identifier> “{“ <member>* “}”**

Each exception is characterized by its OMG IDL identifier, an exception type identifier, and the type of the associated return value (as specified by the **<member>** in its declaration). If an exception is returned as the outcome to a request, then the value of the exception identifier is accessible to the programmer for determining which particular exception was raised.

If an exception is declared with members, a programmer will be able to access the values of those members when an exception is raised. If no members are specified, no additional information is accessible when an exception is raised.

A set of standard exceptions is defined corresponding to standard run-time errors which may occur during the execution of a request. These standard exceptions are documented in Section 3.17, “Standard Exceptions,” on page 3-51.

3.12 Operation Declaration

Operation declarations in OMG IDL are similar to C function declarations. The syntax is:

<op_dcl> ::= [<op_attribute>] <op_type_spec> <identifier>
<parameter_dcls> [<raises_expr>]
[<context_expr>]

<op_type_spec> ::= <param_type_spec>
| “void”

An operation declaration consists of:

- An optional operation attribute that specifies which invocation semantics the communication system should provide when the operation is invoked. Operation attributes are described in Section 3.12.1, “Operation Attribute,” on page 3-42.
- The type of the operation’s return result; the type may be any type which can be defined in OMG IDL. Operations that do not return a result must specify the **void** type.
- An identifier that names the operation in the scope of the interface in which it is defined.
- A parameter list that specifies zero or more parameter declarations for the operation. Parameter declaration is described in Section 3.12.2, “Parameter Declarations,” on page 3-42.
- An optional raises expression which indicates which exceptions may be raised as a result of an invocation of this operation. Raises expressions are described in Section 3.12.3, “Raises Expressions,” on page 3-43.
- An optional context expression which indicates which elements of the request context may be consulted by the method that implements the operation. Context expressions are described in Section 3.12.4, “Context Expressions,” on page 3-43.

Some implementations and/or language mappings may require operation-specific pragmas to immediately precede the affected operation declaration.

3.12.1 Operation Attribute

The operation attribute specifies which invocation semantics the communication service must provide for invocations of a particular operation. An operation attribute is optional. The syntax for its specification is as follows:

<op_attribute>::=“oneway”

When a client invokes an operation with the **oneway** attribute, the invocation semantics are best-effort, which does not guarantee delivery of the call; best-effort implies that the operation will be invoked at most once. An operation with the **oneway** attribute must not contain any output parameters and must specify a **void** return type. An operation defined with the **oneway** attribute may not include a raises expression; invocation of such an operation, however, may raise a standard exception.

If an **<op_attribute>** is not specified, the invocation semantics is at-most-once if an exception is raised; the semantics are exactly-once if the operation invocation returns successfully.

3.12.2 Parameter Declarations

Parameter declarations in OMG IDL operation declarations have the following syntax:

```
<parameter_dcls>    ::= “(” <param_dcl> { “,” <param_dcl> }* “)”
                        | “(” “)”

<param_dcl>         ::= <param_attribute> <param_type_spec>
                        <simple_declarator>

<param_attribute>   ::= “in”
                        | “out”
                        | “inout”

<param_type_spec>   ::= <base_type_spec>
                        | <string_type>
                        | <scoped_name>
```

A parameter declaration must have a directional attribute that informs the communication service in both the client and the server of the direction in which the parameter is to be passed. The directional attributes are:

- **in** - the parameter is passed from client to server.
- **out** - the parameter is passed from server to client.
- **inout** - the parameter is passed in both directions.

It is expected that an implementation will *not* attempt to modify an **in** parameter. The ability to even attempt to do so is language-mapping specific; the effect of such an action is undefined.

If an exception is raised as a result of an invocation, the values of the return result and any **out** and **inout** parameters are undefined.

3.12.3 Raises Expressions

A **raises** expression specifies which exceptions may be raised as a result of an invocation of the operation. The syntax for its specification is as follows:

<raises_expr>::=“raises” “(” <scoped_name> { “,” <scoped_name> }* “)”

The **<scoped_name>**s in the **raises** expression must be previously defined exceptions.

In addition to any operation-specific exceptions specified in the **raises** expression, there are a standard set of exceptions that may be signalled by the ORB. These standard exceptions are described in Section 3.17, “Standard Exceptions,” on page 3-51. However, standard exceptions may *not* be listed in a **raises** expression.

The absence of a **raises** expression on an operation implies that there are no operation-specific exceptions. Invocations of such an operation are still liable to receive one of the standard exceptions.

3.12.4 Context Expressions

A **context** expression specifies which elements of the client’s context may affect the performance of a request by the object. The syntax for its specification is as follows:

<context_expr>::=“context” “(” <string_literal> { “,” <string_literal> }* “)”

The run-time system guarantees to make the value (if any) associated with each **<string_literal>** in the client’s context available to the object implementation when the request is delivered. The ORB and/or object is free to use information in this *request context* during request resolution and performance.

The absence of a context expression indicates that there is no request context associated with requests for this operation.

Each **string_literal** is an arbitrarily long sequence of alphabetic, digit, period (“.”), underscore (“_”), and asterisk (“*”) characters. The first character of the string must be an alphabetic character. An asterisk may only be used as the last character of the string. Some implementations may use the period character to partition the name space.

The mechanism by which a client associates values with the context identifiers is described in the Dynamic Invocation Interface chapter.

3.13 Attribute Declaration

An interface can have attributes as well as operations; as such, attributes are defined as part of an interface. An attribute definition is logically equivalent to declaring a pair of accessor functions; one to retrieve the value of the attribute and one to set the value of the attribute.

The syntax for **attribute** declaration is:

```
<attr_dcl>      ::= [ "readonly" ] "attribute" <param_type_spec>
                    <simple_declarator>
                    { "," <simple_declarator> }*
```

The optional **readonly** keyword indicates that there is only a single accessor function—the retrieve value function. Consider the following example:

```
interface foo {
    enum material_t {rubber, glass};
    struct position_t {
        float x, y;
    };

    attribute float radius;
    attribute material_t material;
    readonly attribute position_t position;

    • • •
};
```

The attribute declarations are equivalent to the following pseudo-specification fragment, assuming that one of the leading '_'s is removed by application of the Escaped Identifier rule described in Section 3.2.3.1, "Escaped Identifiers," on page 3-7:

```
• • •
float       __get_radius ();
void        __set_radius (in float r);
material_t  __get_material ();
void        __set_material (in material_t m);
position_t  __get_position ();
• • •
```

The actual accessor function names are language-mapping specific. The attribute name is subject to OMG IDL's name scoping rules; the accessor function names are guaranteed *not* to collide with any legal operation names specifiable in OMG IDL.

Attribute operations return errors by means of standard exceptions.

Attributes are inherited. An attribute name *cannot* be redefined to be a different type. See Section 3.14, "CORBA Module," on page 3-44 for more information on redefinition constraints and the handling of ambiguity.

3.14 CORBA Module

Names defined by the CORBA specification are in a module named CORBA. In an OMG IDL specification, however, OMG IDL keywords such as **Object** must not be preceded by a "**CORBA::**" prefix. Other interface names such as TypeCode are not OMG IDL keywords, so they must be referred to by their fully scoped names (e.g., **CORBA::TypeCode**) within an OMG IDL specification.

For example in:

```
#include <orb.idl>
module M {
    typedef CORBA::Object myObjRef;     // Error: keyword Object scoped
    typedef TypeCode myTypeCode;        // Error: TypeCode undefined
    typedef CORBA::TypeCode TypeCode;// OK

};
```

The file **orb.idl** contains the IDL definitions for the CORBA module. The file **orb.idl** must be included in IDL files that use names defined in the CORBA module.

The version of **CORBA** specified in this release of the specification is version **<x.y>**, and this is reflected in the IDL for the **CORBA** module by including the following pragma version (see Section 10.6.5.3, “The Version Pragma,” on page 10-45):

```
#pragma version CORBA <x.y>
```

as the first line immediately following the very first **CORBA** module introduction line, which in effect associates that version number with the **CORBA** entry in the **IR**. The version number in that version pragma line must be changed whenever any changes are made to any remotely accessible parts of the **CORBA** module in an officially released OMG standard.

3.15 Names and Scoping

OMG IDL identifiers are case insensitive; that is, two identifiers that differ only in the case of their characters are considered redefinitions of one another. However, all references to a definition must use the same case as the defining occurrence. This allows natural mappings to case-sensitive languages. So for example:

```
module M {
    typedef long Long;      // Error: Long clashes with keyword long
    typedef long TheThing;
    interface I {
        typedef long MyLong;
        myLong op1(                 // Error: inconsistent capitalization
            in TheThing thething;  // Error: TheThing clashes with thething
        );
    };
};
```

3.15.1 Qualified Names

A qualified name (one of the form <scoped-name>::<identifier>) is resolved by first resolving the qualifier <scoped-name> to a scope S, and then locating the definition of <identifier> within S. The identifier must be directly defined in S or (if S is an interface) inherited into S. The <identifier> is not searched for in enclosing scopes.

When a qualified name begins with “::”, the resolution process starts with the file scope and locates subsequent identifiers in the qualified name by the rule described in the previous paragraph.

Every OMG IDL definition in a file has a global name within that file. The global name for a definition is constructed as follows.

Prior to starting to scan a file containing an OMG IDL specification, the name of the current root is initially empty (“”) and the name of the current scope is initially empty (“”). Whenever a **module** keyword is encountered, the string “::” and the associated identifier are appended to the name of the current root; upon detection of the termination of the **module**, the trailing “::” and identifier are deleted from the name of the current root. Whenever an **interface**, **struct**, **union**, or **exception** keyword is encountered, the string “::” and the associated identifier are appended to the name of the current scope; upon detection of the termination of the **interface**, **struct**, **union**, or **exception**, the trailing “::” and identifier are deleted from the name of the current scope. Additionally, a new, unnamed, scope is entered when the parameters of an operation declaration are processed; this allows the parameter names to duplicate other identifiers; when parameter processing has completed, the unnamed scope is exited.

The global name of an OMG IDL definition is the concatenation of the current root, the current scope, a “::”, and the <identifier>, which is the local name for that definition.

Note that the global name in an OMG IDL files corresponds to an absolute **ScopedName** in the Interface Repository. (See Section 10.5.1, “Supporting Type Definitions,” on page 10-10).

Inheritance produces shadow copies of the inherited identifiers; that is, it introduces names into the derived interface, but these names are considered to be semantically the same as the original definition. Two shadow copies of the same original (as results from the diamond shape in Figure 3-1 on page 3-21) introduce a single name into the derived interface and don’t conflict with each other.

Inheritance introduces multiple global OMG IDL names for the inherited identifiers. Consider the following example:

```
interface A {
    exception E {
        long L;
    };
    void f() raises(E);
};

interface B: A {
    void g() raises(E);
};
```

In this example, the exception is known by the global names **::A::E** and **::B::E**.

Ambiguity can arise in specifications due to the nested naming scopes. For example:

```
interface A {
    typedef string<128> string_t;
};

interface B {
    typedef string<256> string_t;
};

interface C: A, B {
    attribute string_t      Title;      // Error: Ambiguous
    attribute A::string_t   Name;       // OK
    attribute B::string_t   City;       // OK
};
```

The declaration of attribute **Title** in interface **C** is ambiguous, since the compiler does not know which **string_t** is desired. Ambiguous declarations yield compilation errors.

3.15.2 Scoping Rules and Name Resolution

Contents of an entire OMG IDL file, together with the contents of any files referenced by #include statements, forms a naming scope. Definitions that do not appear inside a scope are part of the global scope. There is only a single global scope, irrespective of the number of source files that form a specification.

The following kinds of definitions form scopes:

- module
- interface
- valuetype
- struct
- union
- operation
- exception

The scope for module, interface, valuetype, struct and exception begins immediately following its opening '{' and ends immediately preceding its closing '}'. The scope of an operation begins immediately following its '(' and ends immediately preceding its closing ')'. The scope of an union begins immediately following the '(' following the keyword **switch**, and ends immediately preceding its closing '}'. The appearance of the declaration of any of these kinds in any scope, subject to semantic validity of such declaration, opens a nested scope associated with that declaration.

An identifier can only be defined once in a scope. However, identifiers can be redefined in nested scopes. An identifier declaring a module is considered to be defined by its first occurrence in a scope. Subsequent occurrences of a module declaration with the same identifier within the same scope reopens the module and hence its scope, allowing additional definitions to be added to it.

The name of an interface, value type, struct, union, exception or a module may not be redefined within the immediate scope of the interface, value type, struct, union, exception, or the module. For example:

```
module M {
    typedef short M;    // Error: M is the name of the module
                        //        in the scope of which the typedef is.
    interface I {
        void i (in short j);    // Error: i clashes with the interface name I
    };
};
```

An identifier from a surrounding scope is introduced into a scope if it is used in that scope. An identifier is not introduced into a scope by merely being visible in that scope. The use of a scoped name introduces the identifier of the outermost scope of the scoped name. For example in:

```
module M {
    module Inner1 {
        typedef string S1;
    };

    module Inner2 {
        typedef string inner1;      // OK
    };
}
```

The declaration of **Inner2::inner1** is OK because the identifier **Inner1**, while visible in module **Inner2**, has not been introduced into module **Inner2** by actual use of it. On the other hand, if module **Inner2** were:

```
module Inner2{
    typedef Inner1::S1 S2;      // Inner1 introduced
    typedef string inner1;      // Error
    typedef string S1;          // OK
};
```

The definition of **inner1** is now an error because the identifier **Inner1** referring to the **module Inner1** has been introduced in the scope of module **Inner2** in the first line of the module declaration. Also, the declaration of **S1** in the last line is OK since the identifier **S1** was not introduced into the scope by the use of **Inner1::S1** in the first line.

Enumeration value names are introduced into the enclosing scope and then are treated like any other declaration in that scope. For example:

```
interface A {
    enum E { E1, E2, E3 };       // line 1

    enum BadE { E3, E4, E5 }; // Error: E3 is already introduced
                              // into the A scope in line 1 above
};

interface C {
    enum AnotherE { E1, E2, E3 };
};

interface D : C, A {
    union U switch ( E ) {
        case A::E1 : boolean b;// OK.
        case E2 : long l;        // Error: E2 is ambiguous (notwithstanding
                                 // the switch type specification!!)
    };
};
```

Type names defined in a scope are available for immediate use within that scope. In particular, see Section 3.10.2, “Constructed Types,” on page 3-35 on cycles in type definitions.

A name can be used in an unqualified form within a particular scope; it will be resolved by successively searching farther out in enclosing scopes, while taking into consideration inheritance relationships among interfaces. For example:

```
module M {
    typedef long ArgType;
    typedef ArgType AType;          // line l1
    interface B {
        typedef string ArgType;     // line l3
        ArgType opb(in AType i);    // line l2
    };
};

module N {
    typedef char ArgType;           // line l4
    interface Y : M::B {
        void opy(in ArgType i);     // line l5
    };
};
```

The following scopes are searched for the declaration of **ArgType** used on **line l5**:

1. Scope of **N::Y** before the use of **ArgType**.

2. Scope of **N::Y**’s base interface **M::B**. (inherited scope)

3. Scope of **module N** before the definition of **N::Y**.

4. Global scope before the definition of **N**.

M::B::ArgType is found in **step 2** in **line l3**, and that is the definition that is used in **line l5**, hence **ArgType** in **line l5** is **string**. It should be noted that **ArgType** is not **char** in **line l5**. Now if **line l3** were removed from the definition of interface **M::B** then **ArgType** on **line l5** would be **char** from **line l4** which is found in **step 3**.

Following analogous search steps for the types used in the operation **M::B::opb** on **line l2**, the type of **AType** used on **line l2** is **long** from the **typedef** in **line l1** and the return type **ArgType** is **string** from **line l3.**

3.15.3 Special Scoping Rules for Type Names

Once a type has been *defined* anywhere within the scope of a module, interface or valuetype, it may not be redefined except within the scope of a nested module or interface. For example:

```
module M {
    typedef long ArgType;
    interface A {
        typedef string ArgType;    // OK, redefined in nested scope
        struct S {
            ArgType x;             // x is a string
        };
    };
    typedef double ArgType;        // Error: redefinition in same scope
};
```

Once a type identifier has been *used* anywhere within the scope of an interface or valuetype, it may not be redefined within the scope of that interface or valuetype. Use of type names within nested scopes created by structs, unions, and exceptions, as well as within the unnamed scope created by an operation parameter list, are for these purposes considered to occur within the scope of the enclosing interface or valuetype. For example:

```
module M {
    typedef long ArgType;
    const long I = 10;
    typedef short Y;

    interface A {
        struct S {
            ArgType x[I];    // x is a long[10], ArgType and I are used
            long y;      // Note: a new y is defined; the existing Y is not used
        };
        typedef string ArgType;    // Error: ArgType redefined after use
        enum I {I1, I2};           // Error: I redefined after use
        typedef short Y;           // OK because Y has not been used yet!
    };
};
```

Note that redefinition of a type after use in a module is OK as in the example:

```
typedef long ArgType;
module M {
    struct S {
        ArgType x;          // x is a long
    };
    typedef string ArgType;    // OK!
    struct T {
        ArgType y;          // Ugly but OK, y is a string
    };
};
```

3.16 Differences from C++

The OMG IDL grammar, while attempting to conform to the C++ syntax, is somewhat more restrictive. The restrictions are as follows:

- A function return type is mandatory.
- A name must be supplied with each formal parameter to an operation declaration.
- A parameter list consisting of the single token **void** is *not* permitted as a synonym for an empty parameter list.
- Tags are required for structures, discriminated unions, and enumerations.
- Integer types cannot be defined as simply int or unsigned; they must be declared explicitly as **short, long** or **long long**.
- **char** cannot be qualified by **signed** or **unsigned** keywords.

3.17 Standard Exceptions

This section presents the standard exceptions defined for the ORB. These exception identifiers may be returned as a result of any operation invocation, regardless of the interface specification. Standard exceptions may not be listed in **raises** expressions.

In order to bound the complexity in handling the standard exceptions, the set of standard exceptions should be kept to a tractable size. This constraint forces the definition of equivalence classes of exceptions rather than enumerating many similar exceptions. For example, an operation invocation can fail at many different points due to the inability to allocate dynamic memory. Rather than enumerate several different exceptions corresponding to the different ways that memory allocation failure causes the exception (during marshaling, unmarshaling, in the client, in the object implementation, allocating network packets), a single exception corresponding to dynamic memory allocation failure is defined.

Each standard exception includes a minor code to designate the subcategory of the exception.

Minor exception codes are of type **unsigned long** and consist of a 20-bit “Vendor Minor Codeset ID”(**VMCID**), which occupies the high order 20 bits, and the minor code which occupies the low order 12 bits.

Minor codes for the standard exceptions are prefaced by the **VMCID** assigned to OMG, defined as the unsigned long constant **CORBA::OMGVMCID**, which has the VMCID allocated to OMG occupying the high order 20 bits. The minor exception codes associated with the standard exceptions that are found in Table 3-13 on page 3-58 are or-ed with **OMGVMCID** to get the minor code value that is returned in the ex_body structure (see Section 3.17.1, “Standard Exception Definitions,” on page 3-52 and Section 3.17.2, “Standard Minor Exception Codes,” on page 3-58).

Within a vendor assigned space, the assignment of values to minor codes is left to the vendor. Vendors may request allocation of **VMCID**s by sending email to tag-*request@omg.org*.

The **VMCID** *0* and *\xfffff* are reserved for experimental use. The **VMCID OMGVMCID** (Section 3.17.1, “Standard Exception Definitions,” on page 3-52) and *1 through \xf* are reserved for OMG use.

Each standard exception also includes a **completion_status** code which takes one of the values {COMPLETED_YES, COMPLETED_NO, COMPLETED_MAYBE}. These have the following meanings:

COMPLETED_YES	The object implementation has completed processing prior to the exception being raised.
COMPLETED_NO	The object implementation was never initiated prior to the exception being raised.
COMPLETED_MAYBE	The status of implementation completion is indeterminate.

3.17.1 Standard Exception Definitions

The standard exceptions are defined below. Clients must be prepared to handle system exceptions that are not on this list, both because future versions of this specification may define additional standard exceptions, and because ORB implementations may raise non-standard system exceptions.

```
module CORBA {
    const unsigned long OMGVMCID = \x4f4d0000;

#define ex_body {unsigned long minor; completion_status completed;}
    enum completion_status {COMPLETED_YES,
                            COMPLETED_NO,
                            COMPLETED_MAYBE};

    enum exception_type {   NO_EXCEPTION,
                            USER_EXCEPTION,
                            SYSTEM_EXCEPTION};

    exception UNKNOWN       ex_body;  // the unknown exception
    exception BAD_PARAM     ex_body;  // an invalid parameter was
                                      // passed
    exception NO_MEMORY     ex_body;  // dynamic memory allocation
```

```
                                                    // failure
    exception IMP_LIMIT             ex_body;        // violated implementation
                                                    //  limit
    exception COMM_FAILURE          ex_body;        // communication failure
    exception INV_OBJREF            ex_body;        // invalid object reference
    exception NO_PERMISSION         ex_body;        // no permission for
                                                    // attempted op.
    exception INTERNAL              ex_body;        // ORB internal error
    exception MARSHAL               ex_body;        // error marshaling
                                                    // param/result

    exception INITIALIZE            ex_body;        // ORB initialization failure
    exception NO_IMPLEMENT          ex_body;        // operation implementation
                                                    // unavailable

    exception BAD_TYPECODE          ex_body;        // bad typecode
    exception BAD_OPERATION         ex_body;        // invalid operation
    exception NO_RESOURCES          ex_body;        // insufficient resources
                                                    // for req.
    exception NO_RESPONSE           ex_body;        // response to req. not yet
                                                    // available
    exception PERSIST_STORE         ex_body;        // persistent storage failure
    exception BAD_INV_ORDER         ex_body;        // routine invocations
                                                    // out of order
    exception TRANSIENT             ex_body;        // transient failure - reissue
                                                    // request

    exception FREE_MEM              ex_body;        // cannot free memory
    exception INV_IDENT             ex_body;        // invalid identifier syntax
    exception INV_FLAG              ex_body;        // invalid flag was specified
    exception INTF_REPOS            ex_body;        // error accessing interface
                                                    // repository
    exception BAD_CONTEXT           ex_body;        // error processing context
                                                    // object
    exception OBJ_ADAPTER           ex_body;        // failure detected by object
                                                    // adapter

    exception DATA_CONVERSION       ex_body;        // data conversion error
    exception OBJECT_NOT_EXIST      ex_body;        // non-existent object,
                                                    // delete reference
    exception TRANSACTION_REQUIRED
                                    ex_body;        // transaction required
    exception TRANSACTION_ROLLEDBACK
                                    ex_body;        // transaction rolled
                                                    // back
    exception INVALID_TRANSACTION
                                    ex_body;        // invalid transaction
    exception INV_POLICY            ex_body;        // invalid policy
    exception CODESET_INCOMPATIBLE
                                    ex_body         // incompatible code set
};
```

3.17.1.1 UNKNOWN

This exception is raised if an operation implementation throws a non-CORBA exception (such as an exception specific to the implementation's programming language), or if an operation raises a user exception that does not appear in the operation's raises expression. UNKNOWN is also raised if the server returns a system exception that is unknown to the client. (This can happen if the server uses a later version of CORBA than the client and new system exceptions have been added to the later version.)

3.17.1.2 BAD_PARAM

A parameter passed to a call is out of range or otherwise considered illegal. An ORB may raise this exception if null values or null pointers are passed to an operation (for language mappings where the concept of a null pointers or null values applies). BAD_PARAM can also be raised as a result of client generating requests with incorrect parameters using the DII.

3.17.1.3 NO_MEMORY

The ORB run time has run out of memory.

3.17.1.4 IMP_LIMIT

This exception indicates that an implementation limit was exceeded in the ORB run time. For example, an ORB may reach the maximum number of references it can hold simultaneously in an address space, the size of a parameter may have exceeded the allowed maximum, or an ORB may impose a maximum on the number of clients or servers that can run simultaneously.

3.17.1.5 COMM_FAILURE

This exception is raised if communication is lost while an operation is in progress, after the request was sent by the client, but before the reply from the server has been returned to the client.

3.17.1.6 INV_OBJREF

This exception indicates that an object reference is internally malformed. For example, the repository ID may have incorrect syntax or the addressing information may be invalid. This exception is raised by **ORB::string_to_object** if the passed string does not decode correctly.

An ORB may choose to detect calls via nil references (but is not obliged to do detect them). INV_OBJREF is used to indicate this.

3.17.1.7 NO_PERMISSION

An invocation failed because the caller has insufficient privileges.

3.17.1.8 INTERNAL

This exception indicates an internal failure in an ORB, for example, if an ORB has detected corruption of its internal data structures.

3.17.1.9 MARSHAL

A request or reply from the network is structurally invalid. This error typically indicates a bug in either the client-side or server-side run time. For example, if a reply from the server indicates that the message contains 1000 bytes, but the actual message is shorter or longer than 1000 bytes, the ORB raises this exception. MARSHAL can also be caused by using the DII or DSI incorrectly, for example, if the type of the actual parameters sent does not agree with IDL signature of an operation.

3.17.1.10 INITIALIZE

An ORB has encountered a failure during its initialization, such as failure to acquire networking resources or detecting a configuration error.

3.17.1.11 NO_IMPLEMENT

This exception indicates that even though the operation that was invoked exists (it has an IDL definition), no implementation for that operation exists. NO_IMPLEMENT can, for example, be raised by an ORB if a client asks for an object's type definition from the interface repository, but no interface repository is provided by the ORB.

3.17.1.12 BAD_TYPECODE

The ORB has encountered a malformed type code (for example, a type code with an invalid **TCKind** value).

3.17.1.13 BAD_OPERATION

This indicates that an object reference denotes an existing object, but that the object does not support the operation that was invoked.

3.17.1.14 NO_RESOURCES

The ORB has encountered some general resource limitation. For example, the run time may have reached the maximum permissible number of open connections.

3.17.1.15 NO_RESPONSE

This exception is raised if a client attempts to retrieve the result of a deferred synchronous call, but the response for the request is not yet available.

3.17.1.16 PERSIST_STORE

This exception indicates a persistent storage failure, for example, failure to establish a database connection or corruption of a database.

3.17.1.17 BAD_INV_ORDER

This exception indicates that the caller has invoked operations in the wrong order. For example, it can be raised by an ORB if an application makes an ORB-related call without having correctly initialized the ORB first.

3.17.1.18 TRANSIENT

TRANSIENT indicates that the ORB attempted to reach an object and failed. It is not an indication that an object does not exist. Instead, it simply means that no further determination of an object's status was possible because it could not be reached. This exception is raised if an attempt to establish a connection fails, for example, because the server or the implementation repository is down.

3.17.1.19 FREE_MEM

The ORB failed in an attempt to free dynamic memory, for example because of heap corruption or memory segments being locked.

3.17.1.20 INV_IDENT

This exception indicates that an IDL identifier is syntactically invalid. It may be raised if, for example, an identifier passed to the interface repository does not conform to IDL identifier syntax, or if an illegal operation name is used with the DII.

3.17.1.21 INV_FLAG

An invalid flag was passed to an operation (for example, when creating a DII request).

3.17.1.22 INTF_REPOS

An ORB raises this exception if it cannot reach the interface repository, or some other failure relating to the interface repository is detected.

3.17.1.23 BAD_CONTEXT

An operation may raise this exception if a client invokes the operation but the passed context does not contain the context values required by the operation.

3.17.1.24 OBJ_ADAPTER

This exception typically indicates an administrative mismatch. For example, a server may have made an attempt to register itself with an implementation repository under a name that is already in use, or is unknown to the repository. OBJ_ADAPTER is also raised by the POA to indicate problems with application-supplied servant managers.

3.17.1.25 DATA_CONVERSION

This exception is raised if an ORB cannot convert the representation of data as marshaled into its native representation or vice-versa. For example, DATA_CONVERSION can be raised if wide character codeset conversion fails, or if an ORB cannot convert floating point values between different representations.

3.17.1.26 OBJECT_NOT_EXIST

The OBJECT_NOT_EXIST exception is raised whenever an invocation on a deleted object was performed. It is an authoritative “hard” fault report. Anyone receiving it is allowed (even expected) to delete all copies of this object reference and to perform other appropriate “final recovery” style procedures.

Bridges forward this exception to clients, also destroying any records they may hold (for example, proxy objects used in reference translation). The clients could in turn purge any of their own data structures.

3.17.1.27 TRANSACTION_REQUIRED

The TRANSACTION_REQUIRED exception indicates that the request carried a null transaction context, but an active transaction is required.

3.17.1.28 TRANSACTION_ROLLEDBACK

The TRANSACTION_ROLLEDBACK exception indicates that the transaction associated with the request has already been rolled back or marked to roll back. Thus, the requested operation either could not be performed or was not performed because further computation on behalf of the transaction would be fruitless.

3.17.1.29 INVALID_TRANSACTION

The INVALID_TRANSACTION indicates that the request carried an invalid transaction context. For example, this exception could be raised if an error occurred when trying to register a resource.

3.17.1.30 INV_POLICY

INV_POLICY is raised when an invocation cannot be made due to an incompatibility between Policy overrides that apply to the particular invocation.

3.17.1.31 CODESET_INCOMPATIBLE

This exception is raised whenever meaningful communication is not possible between client and server native code sets. See Section 13.7.2.6, “Code Set Negotiation,” on page 13-34.

3.17.2 Standard Minor Exception Codes

The following table specifies standard minor exception codes that have been assigned for the standard exceptions. The actual value that is to be found in the **minor** field of the **ex_body** structure is obtained by or-ing the values in this table with the **OMGVMCID** constant. For example “Missing local value implementation” for the exception NO_IMPLEMENT would be denoted by the **minor** value **\x4f4d0001**.

Table 3-13 Minor Exception Codes

SYSTEM EXCEPTION	MINOR CODE	EXPLANATION
BAD_PARAM	1	Failure to register, unregister or lookup value factory
	2	RID already defined in IFR
	3	Name already used in the context in IFR
	4	Target is not a valid container
	5	Name clash in inherited context
	6	Incorrect type for abstract interface
MARSHAL	1	Unable to locate value factory
NO_IMPLEMENT	1	Missing local value implementation
	2	Incompatible value implementation version
BAD_INV_ORDER	1	Dependency exists in IFR preventing destruction of this object
	2	Attempt to destroy indestructible objects in IFR
	3	Operation would deadlock
	4	ORB has shutdown
OBJECT_NOT_EXIST	1	Attempt to pass an unactivated (unregistered) value as an object reference