OMG IDL Syntax and Semantics 3

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 "Lexical Conventions" on page 3-3. A description of OMG IDL preprocessing is presented in "Preprocessing" on page 3-9. The scope rules for identifiers in an OMG IDL specification are described in "CORBA Module" on page 3-34.

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 "OMG IDL Grammar" on page 3-10.

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.

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.

---

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

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 ISO Latin-1 (8859.1) character set. This character set is divided into alphabetic characters (letters), digits, graphic characters, the space (blank) character and formatting characters. Table 3-2 shows the OMG IDL alphabetic characters; upper- and lower-case equivalencies are paired.

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

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

Table 3-2 The 114 Alphabetic Characters (Letters) *(Continued)*

Char.	Description	Char.	Description
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

Table 3-4 The 65 Graphic Characters *(Continued)*

Char.	Description	Char.	Description
'	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		vulgar fraction 1/4
{	left curly bracket		vulgar fraction 1/2
|	vertical line		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 alphabetic, digit, and underscore (“_”) characters. The first character must be an alphabetic character. All characters are significant.

Identifiers that differ only in case collide and yield a compilation error. An identifier for a definition must be spelled consistently (with respect to case) throughout a specification.

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 3-3 defines the equivalence mapping of upper- and lower-case letters.
- The comparison does *not* take into account equivalences between digraphs and pairs of letters (e.g., “æ” and “ae” are not considered equivalent) or equivalences between accented and non-accented letters (e.g., “Á” and “A” are not considered equivalent).
- All characters are significant.

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

3.2.4 Keywords

The identifiers listed in Table 3-6 are reserved for use as keywords and may not be used otherwise.

Table 3-6 Keywords

any	double	interface	readonly	unsigned
attribute	enum	long	sequence	union
boolean	exception	module	short	void
case	FALSE	Object	string	wchar
char	fixed	octet	struct	wstring
const	float	oneway	switch	
context	in	out	TRUE	
default	inout	raises	typedef	

Keywords obey the rules for identifiers (see“Identifiers” on page 3-6) and must be written exactly as shown in the above list. For example, “**boolean**” is correct; “**Boolean**” produces a compilation error. The keyword “**Object**” can be used as a type specifier.

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

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.

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 3-3, Table 3-3 on page 3-4, and Table 3-4 on page 3-4). 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 3-5). 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

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. 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 and wide string literals are specified exactly like character and string literals. All character and string literals, both wide and non-wide, may only be specified (portably) using the characters found in the ISO 8859-1 character set, that is interface names, operation names, type names, etc., will continue to be limited to the ISO 8859-1 character set.

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.

String Literals

A string literal is a sequence of characters (as defined in “Character Literals” on page 3-8) 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'.

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 “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 8.6, “RepositoryIds,” on page 8-32.

3.4 OMG IDL Grammar

```
(1)   <specification>      ::= <definition>+
(2)   <definition>         ::= <type_dcl> ";"
                           |   <const_dcl> ";"
                           |   <except_dcl> ";"
                           |   <interface> ";"
                           |   <module> ";"
(3)   <module>             ::= "module" <identifier> "{" <definition>+ "}"
(4)   <interface>          ::= <interface_dcl>
                           |   <forward_dcl>
(5)   <interface_dcl>      ::= <interface_header> "{" <interface_body> "}"
(6)   <forward_dcl>        ::= "interface" <identifier>
(7)   <interface_header>   ::= "interface" <identifier> [ <inheritance_spec> ]
(8)   <interface_body>     ::= <export>*
(9)   <export>             ::= <type_dcl> ";"
                           |   <const_dcl> ";"
                           |   <except_dcl> ";"
                           |   <attr_dcl> ";"
                           |   <op_dcl> ";"
(10)  <inheritance_spec>   ::= ":" <scoped_name> { "," <scoped_name> }*
```

```
(11)   <scoped_name>    ::= <identifier>
                        |   "::" <identifier>
                        |   <scoped_name> "::" <identifier>
(12)   <const_dcl>      ::= "const" <const_type> <identifier> "="
                            <const_exp>
(13)   <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>
(14)   <const_exp>      ::= <or_expr>
(15)   <or_expr>        ::= <xor_expr>
                        |   <or_expr> "|" <xor_expr>
(16)   <xor_expr>       ::= <and_expr>
                        |   <xor_expr> "^" <and_expr>
(17)   <and_expr>       ::= <shift_expr>
                        |   <and_expr> "&" <shift_expr>
(18)   <shift_expr>     ::= <add_expr>
                        |   <shift_expr> ">>" <add_expr>
                        |   <shift_expr> "<<" <add_expr>
(19)   <add_expr>       ::= <mult_expr>
                        |   <add_expr> "+" <mult_expr>
                        |   <add_expr> "-" <mult_expr>
(20)   <mult_expr>      ::= <unary_expr>
                        |   <mult_expr> "*" <unary_expr>
                        |   <mult_expr> "/" <unary_expr>
                        |   <mult_expr> "%" <unary_expr>
(21)   <unary_expr>     ::= <unary_operator> <primary_expr>
                        |   <primary_expr>
(22)   <unary_operator> ::= "-"
                        |   "+"
                        |   "~"
(23)   <primary_expr>   ::= <scoped_name>
                        |   <literal>
                        |   "(" <const_exp> ")"
(24)   <literal>        ::= <integer_literal>
                        |   <string_literal>
                        |   <wide_string_literal>
                        |   <character_literal>
                        |   <wide_character_literal>
                        |   <fixed_pt_literal>
                        |   <floating_pt_literal>
                        |   <boolean_literal>
```

(25) <boolean_literal> ::= "TRUE"
| "FALSE"
(26) <positive_int_const> ::= <const_exp>
(27) <type_dcl> ::= "typedef" <type_declarator>
| <struct_type>
| <union_type>
| <enum_type>
| "native" <simple_declarator>
(28) <type_declarator> ::= <type_spec> <declarators>
(29) <type_spec> ::= <simple_type_spec>
| <constr_type_spec>
(30) <simple_type_spec> ::= <base_type_spec>
| <template_type_spec>
| <scoped_name>
(31) <base_type_spec> ::= <floating_pt_type>
| <integer_type>
| <char_type>
| <wide_char_type>
| <boolean_type>
| <octet_type>
| <any_type>
| <object_type>
(32) <template_type_spec> ::= <sequence_type>
| <string_type>
| <wide_string_type>
| <fixed_pt_type>
(33) <constr_type_spec> ::= <struct_type>
| <union_type>
| <enum_type>
(34) <declarators> ::= <declarator> { "," <declarator> }*
(35) <declarator> ::= <simple_declarator>
| <complex_declarator>
(36) <simple_declarator> ::= <identifier>
(37) <complex_declarator> ::= <array_declarator>
(38) <floating_pt_type> ::= "float"
| "double"
| "long" "double"
(39) <integer_type> ::= <signed_int>
| <unsigned_int>
(40) <signed_int> ::= <signed_short_int>
| <signed_long_int>
| <signed_longlong_int>
(41) <signed_short_int> ::= "short"
(42) <signed_long_int> ::= "long"
(43) <signed_longlong_int> ::= "long" "long"
(44) <unsigned_int> ::= <unsigned_short_int>
| <unsigned_long_int>

```
                                    |   <unsigned_longlong_int>
(45)      <unsigned_short_int>      ::= "unsigned" "short"
(46)      <unsigned_long_int>       ::= "unsigned" "long"
(47) <unsigned_longlong_int>        ::= "unsigned" "long" "long"
(48)           <char_type>          ::= "char"
(49)      <wide_char_type>          ::= "wchar"
(50)        <boolean_type>          ::= "boolean"
(51)          <octet_type>          ::= "octet"
(52)            <any_type>          ::= "any"
(53)         <object_type>          ::= "Object"
(54)         <struct_type>          ::= "struct" <identifier> "{" <member_list> "}"
(55)         <member_list>          ::= <member>+
(56)             <member>           ::= <type_spec> <declarators> ";"
(57)         <union_type>           ::= "union" <identifier> "switch" "("
                                        <switch_type_spec> ")" "{" <switch_body>
                                        "}"
(58)    <switch_type_spec>          ::= <integer_type>
                                    |   <char_type>
                                    |   <boolean_type>
                                    |   <enum_type>
                                    |   <scoped_name>
(59)        <switch_body>           ::= <case>+
(60)             <case>             ::= <case_label>+ <element_spec> ";"
(61)        <case_label>            ::= "case" <const_exp> ":"
                                    |   "default" ":"
(62)       <element_spec>           ::= <type_spec> <declarator>
(63)         <enum_type>            ::= "enum" <identifier> "{" <enumerator> { ","
                                        <enumerator> }* "}"
(64)         <enumerator>           ::= <identifier>
(65)      <sequence_type>           ::= "sequence" "<" <simple_type_spec> ","
                                        <positive_int_const> ">"
                                    |   "sequence" "<" <simple_type_spec> ">"
(66)        <string_type>           ::= "string" "<" <positive_int_const> ">"
                                    |   "string"
(67)     <wide_string_type>         ::= "wstring" "<" <positive_int_const> ">"
                                    |   "wstring"
(68)     <array_declarator>         ::= <identifier> <fixed_array_size>+
(69)     <fixed_array_size>         ::= "[" <positive_int_const> "]"
(70)           <attr_dcl>           ::= [ "readonly" ] "attribute"
                                        <param_type_spec> <simple_declarator> {
                                        "," <simple_declarator> }*
(71)         <except_dcl>           ::= "exception" <identifier> "{" <member>* "}"
(72)           <op_dcl>             ::= [ <op_attribute> ] <op_type_spec> <identi-
                                        fier> <parameter_dcls> [ <raises_expr> ] [
                                        <context_expr> ]
(73)       <op_attribute>           ::= "oneway"
```

```
(74)          <op_type_spec>  ::= <param_type_spec>
                                |  “void”
(75)          <parameter_dcls>  ::= “(” <param_dcl> { “,” <param_dcl> }* “)”
                                |  “(” “)”
(76)          <param_dcl>  ::= <param_attribute> <param_type_spec>
                               <simple_declarator>
(77)          <param_attribute>  ::= “in”
                                |  “out”
                                |  “inout”
(78)          <raises_expr>  ::= “raises” “(” <scoped_name> { “,”
                               <scoped_name> }* “)”
(79)          <context_expr>  ::= “context” “(” <string_literal> { “,”
                               <string_literal> }* “)”
(80)          <param_type_spec>  ::= <base_type_spec>
                                |  <string_type>
                                |  <wide_string_type>
                                |  <fixed_pt_type>
                                |  <scoped_name>
(81)          <fixed_pt_type>  ::= “fixed” “<“ <positive_int_const> “,”
                               <integer_literal> “>”
(82)      <fixed_pt_const_type>  ::= “fixed”
```

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> “;”
```

See “Constant Declaration” on page 3-18, “Type Declaration” on page 3-22, and “Exception Declaration” on page 3-30, respectively, for specifications of **<const_dcl>**, **<type_dcl>**, and **<except_dcl>**.

3.5.1 Module Declaration

A module definition satisfies the following syntax:

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

The module construct is used to scope OMG IDL identifiers; see “CORBA Module” on page 3-34 for details.

3.5.2 Interface Declaration

An interface definition satisfies the following syntax:

```
<interface>          ::= <interface_dcl>
                     |   <forward_dcl>
<interface_dcl>      ::= <interface_header> "{" <interface_body> "}"
<forward_dcl>        ::= "interface" <identifier>
<interface_header>   ::= "interface" <identifier> [<inheritance_spec> ]
<interface_body>     ::= <export>*
<export>             ::= <type_dcl> ";"
                     |   <const_dcl> ";"
                     |   <except_dcl> ";"
                     |   <attr_dcl> ";"
                     |   <op_dcl> ";"
```

Interface Header

The interface header consists of two elements:

- 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.

Inheritance Specification

The syntax for inheritance is as follows:

```
<inheritance_spec>::= ":" <scoped_name> {"," <scoped_name>}*
<scoped_name>        ::= <identifier>
                     | "::" <identifier>
                     | <scoped_name> "::" <identifier>
```

Each **<scoped_name>** in an **<inheritance_spec>** must denote a previously defined interface. See "Inheritance" on page 3-16 for the description of inheritance.

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 “Constant Declaration” on page 3-18.
- Type declarations, which specify the type definitions that the interface exports; type declaration syntax is described in “Type Declaration” on page 3-22.
- Exception declarations, which specify the exception structures that the interface exports; exception declaration syntax is described in “Exception Declaration” on page 3-30.
- Attribute declarations, which specify the associated attributes exported by the interface; attribute declaration syntax is described in “Attribute Declaration” on page 3-33.
- 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 “Operation Declaration” on page 3-31.

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

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

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.

3.6 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 “CORBA Module” on page 3-34.

An interface is called a direct base if it is mentioned in the **<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 **<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 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 { ... }
```

The relationships between these interfaces is shown in Figure on page 3-17. This “diamond” shape is legal.

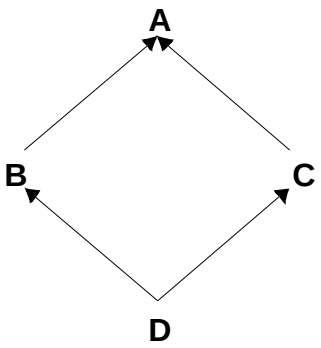

Figure 3-1 Legal Multiple Inheritance Example

Reference to base interface elements must be unambiguous. Reference to a base interface element is ambiguous if the expression used refers to a constant, type, or exception in more than one base interface. (It is currently 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.) Ambiguities can be resolved by qualifying a name with its interface name (that is, using a **<scoped_name>**).

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 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 is a compilation error.

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

3.7 Constant Declaration

This section describes the syntax for constant declarations.

3.7.1 Syntax

The syntax for a constant declaration is:

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

<const_type>      ::= <integer_type>
                  |   <char_type>
                  |   <boolean_type>
                  |   <floating_pt_type>
```

```
                    |   <string_type>
                    |   <scoped_name>

<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>
```

3.7.2 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>**, **<fixed_pt_const_type>**, **<string_type>,** or **<wide_string_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 **3000.00** 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 $\mathbf{s}_{inf}$. The computation proceeds pairwise, with the usual rules for left-to-right association, operator precedence, and parentheses. 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.

3.8 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>

<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.

3.8.1 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 "Standard Exceptions" on page 3-37.

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 3-10.

Table 3-10 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	$0 .. 2^{16} - 1$
unsigned long	$0 .. 2^{32} - 1$
unsigned long long	$0 .. 2^{64} - 1$

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.

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 3-3, Table 3-3 on page 3-4, and Table 3-4 on page 3-4). The meaning and representation of the null and formatting characters (see Table 3-5 on page 3-5) 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.

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.

Boolean Type

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

Octet Type

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

Any Type

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

3.8.2 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 “Sequences” on page 3-27 for details of the **sequence** template type.

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.

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 3-11.

Table 3-11 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.

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.

3.8.3 Template Types

The template types are:

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

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.

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.

Wide Char String Type

The **wstring** data type represents a null-terminated (note: a wide character null) sequence of **wchar**. Type **wstring** is analogous to **string**, except that its element type is **wchar** instead of **char**.

Fixed Type

The **fixed** data type represents a fixed-point decimal number of up to 31 significant digits. The scale factor is normally 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.). However, some languages and environments may be able to accommodate types that have a negative scale or a scale greater than the number of digits.

3.8.4 Complex Declarator

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.

3.8.5 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 com

---

3.9 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 “Standard Exceptions” on page 3-37.

3.10 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 “Operation Attribute” on page 3-31.
- 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 “Parameter Declarations” on page 3-32.
- 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 “Raises Expressions” on page 3-32.
- 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 “Context Expressions” on page 3-33.

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

3.10.1 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.

3.10.2 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.

When an unbounded **string** or **sequence** is passed as an **inout** parameter, the returned value cannot be longer than the input value.

3.10.3 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 "Standard Exceptions" on page 3-37. 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.

3.10.4 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.

3.11 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:

```
• • •
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 C, C++, and Smalltalk mappings are described in separate chapters. 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 "CORBA Module" on page 3-34 for more information on redefinition constraints and the handling of ambiguity.

3.12 CORBA Module

In order to prevent names defined in the *CORBA* specification from clashing with names in programming languages and other software systems, all names defined in *CORBA* are treated as if they were defined within 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.

3.13 Names and Scoping

An entire OMG IDL file forms a naming scope. In addition, the following kinds of definitions form nested scopes:

- module
- interface
- structure
- union
- operation
- exception

Identifiers for the following kinds of definitions are scoped:

- types
- constants
- enumeration values
- exceptions
- interfaces
- attributes
- operations

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 within the same scope reopen the module allowing additional definitions to be added to it.

Due to possible restrictions imposed by future language bindings, 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.)

Type names defined in a scope are available for immediate use within that scope. In particular, see “Constructed Types” on page 3-25 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. Once an unqualified name is used in a scope, it cannot be redefined (i.e., if one has used a name defined in an enclosing scope in the current scope, one cannot then redefine a version of the name in the current scope). Such redefinitions yield a compilation error.

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 “Supporting Type Definitions” on page 8-9).

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 3-17) 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;/* AMBIGUOUS!!! */
};
```

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

3.14 Differences from C++

The OMG IDL grammar, while attempting to conform to the C++ syntax, is somewhat more restrictive. The current 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** or **long**.
- **char** cannot be qualified by **signed** or **unsigned** keywords.

3.15 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 marshalling, unmarshalling, 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; the assignment of values to the minor codes is left to each ORB implementation.

Each standard exception also includes a **completion_status** code which takes one of the 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.

3.15.1 Standard Exceptions 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.

```
#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 marshalling 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
```

3.15.2 Object Non-Existence

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.

3.5.3 Transaction Exceptions

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

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.

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.