Date: February 2013

Interface Definition Language

Version 3.5 - Beta 1

OMG Document Number: ptc/2013-02-02
Standard document URL: http://www.omg.org/spec/IDL/3.5

This OMG document replaces the submission document (mars/11-09-08, Alpha). It is an OMG Adopted Beta Specification and is currently in the finalization phase. Comments on the content of this document are welcome, and should be directed to issues@omg.org by September 15, 2012.

You may view the pending issues for this specification from the OMG revision issues web page http://www.omg.org/issues.

The FTF Recommendation and Report for this specification will be published on June 28, 2013. If you are reading this after that date, please download the available specification from the OMG Specifications web page http://www.omg.org/spec/.

OMG’s Issue Reporting Procedure

All OMG specifications are subject to continuous review and improvement. As part of this process we encourage readers to report any ambiguities, inconsistencies, or inaccuracies they may find by completing the Issue Reporting Form listed on the main web page *http://www.omg.org*, under Documents, Report a Bug/Issue (*http://www.omg.org/report_issue.htm*).

Table of Contents

Preface

About the Object Management Group

OMG

Founded in 1989, the Object Management Group, Inc. (OMG) is an open membership, not-for-profit computer industry standards consortium that produces and maintains computer industry specifications for interoperable, portable, and reusable enterprise applications in distributed, heterogeneous environments. Membership includes Information Technology vendors, end users, government agencies, and academia.

OMG member companies write, adopt, and maintain its specifications following a mature, open process. OMG's specifications implement the Model Driven Architecture® (MDA®), maximizing ROI through a full-lifecycle approach to enterprise integration that covers multiple operating systems, programming languages, middleware and networking infrastructures, and software development environments. OMG's specifications include: UML® (Unified Modeling Language™); CORBA® (Common Object Request Broker Architecture); CWM™ (Common Warehouse Metamodel); and industry-specific standards for dozens of vertical markets.

More information on the OMG is available at *http://www.omg.org/*.

OMG Specifications

As noted, OMG specifications address middleware, modeling, and vertical domain frameworks. All OMG specifications are available from the OMG website at:

http://www.omg.org/spec

Specifications are organized by the following categories:

Business Modeling Specifications

Middleware Specifications

- **CORBA/IIOP**
- **Data Distribution Services**
- **Specialized CORBA**

IDL/Language Mapping Specifications

Modeling and Metadata Specifications

- **UML, MOF, CWM, XMI**
- **UML Profile**

Modernization Specifications

Platform Independent Model (PIM), Platform Specific Model (PSM), Interface Specifications

- **CORBAServices**
- **CORBAFacilities**

OMG Domain Specifications

CORBA Embedded Intelligence Specifications

CORBA Security Specifications

All of OMG's formal specifications may be downloaded without charge from our website. (Products implementing OMG specifications are available from individual suppliers.) Copies of specifications, available in PostScript and PDF format, may be obtained from the link cited above or by contacting the Object Management Group, Inc. at:

OMG Headquarters
109 Highland Avenue
Needham, MA 02494
USA
Tel: +1-781-444-0404
Fax: +1-781-444-0320
Email: *pubs@omg.org*

Certain OMG specifications are also available as ISO standards. Please consult *http://www.iso.org*

Typographical Conventions

The type styles shown below are used in this document to distinguish programming statements from ordinary English. However, these conventions are not used in tables or section headings where no distinction is necessary.

Times/Times New Roman - 10 pt.: Standard body text

Helvetica/Arial - 10 pt. Bold: OMG Interface Definition Language (OMG IDL) and syntax elements.

`Courier - 10 pt. Bold:` Programming language elements.

Helvetica/Arial - 10 pt: Exceptions

Note – Terms that appear in *italics* are defined in the glossary. Italic text also represents the name of a document, specification, or other publication.

Issues

The reader is encouraged to report any technical or editing issues/problems with this specification to *http://www.omg.org/report_issue.htm*.

1 Scope

This document specifies the OMG Interface Definition Language (IDL). The IDL was formerly specified as a collection of chapters within the CORBA 3 specification. This document separates the IDL from the CORBA so that specifications that use the IDL can reference it more easily. Having a separate IDL specification document will also allow it to evolve separately from the CORBA specification.

This document was created from the following OMG specification sections:

- CORBA 3.2 specification part 1 ptc/2011-02-03:
  - Chapter 7 “OMG IDL Syntax and Semantics.” Now chapter 5 of this specification.
  - Chapter 9 “Value Type Semantics.” Now chapter 6 of this specification.
  - Chapter 10 “Abstract Interface Semantics.” Now chapter 7 of this specification.”
- CORBA 3.2 specification part 3 ptc/2011-01-16 sections:
  - Section 7.3 “IDL 3+ Grammar.” Inserted as section 5.19 of this specification

This document reflects what was known informally as IDL 3+. There are no changes in this document relative to referenced specification chapters.

2 Conformance Criteria

This document defines the IDL for other specificatins to reference and contains no independent compliance points. It is up to the specifications that depend on this document to define their own conformance criteria.

The IDL Language Mappings are defined in separate documents; each language mapping is its own separate OMG specification.

3 Normative References

The following referenced documents are indispensable for the application of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments)

- ITU-T Recommendation X.902 (1995) | ISO/IEC 10746-2:1996, Information Technology - Open Distributed Processing - Reference Model: Foundations
- ITU-T Recommendation X.903 (1995) | ISO/IEC 10746-3:1996, Information Technology - Open Distributed Processing - Reference Model: Architecture
- ITU-T Recommendation X.920 (1997) | ISO/IEC 14750:1997, Information Technology - Open Distributed Processing - Interface Definition Language
- ISO/IEC 14882:2003, Information Technology - Programming languages - C++
- ISO/IEC 9899:1999, Information Technology - Programming languages - C

- [OMA] Object Management Group, "Object Management Architecture Guide, revision 3.0" , available from http://www.omg.org/oma/
- [RFC2119] IETF RFC 2119, "Key words for use in RFCs to Indicate Requirement Levels", S. Bradner, March 1997. Available from http://ietf.org/rfc/rfc2119

4 Additional Information

4.1 Outline of Contents

This International Standard specification consists of the following:

1. The syntax and semantics of the OMG interface definition language (OMG IDL). Throughout this specification the abbreviation IDL is used, for brevity, as shorthand for OMG IDL.

4.2 Keywords for Requirement Statements

The keywords “must,” “must not,” “shall,” “shall not,” “should,” “should not,” and “may” in this specification are to be interpreted as described in [RFC 2119].

5 IDL Syntax and Semantics

5.1 Overview

This clause describes OMG Interface Definition Language (IDL) semantics and gives the syntax for IDL grammatical constructs.

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 IDL completely defines the interface and fully specifies each operation's parameters. An IDL interface provides the information needed to develop clients that use the interface's operations.

Clients are not written in IDL, which is purely a descriptive language, but in languages for which mappings from IDL concepts have been defined. The mapping of an IDL concept to a client language construct will depend on the facilities available in the client language. For example, an 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 IDL concepts to several programming languages is described in separate specifications.

The description of IDL's lexical conventions is presented in 5.2, Lexical Conventions. A description of IDL preprocessing is presented in 5.3, Preprocessing. The scope rules for identifiers in an IDL specification are described in 5.21, Names and Scoping.

IDL is a declarative language. The grammar is presented in IDL Grammar on page 12 and associated semantics is described in the rest of this clause either in place or through references to other sub clauses of this standard.

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 IDL must have a ".idl" extension.

The description of IDL grammar uses a syntax notation that is similar to Extended Backus-Naur Format (EBNF). Table 5.1 lists the symbols used in this format and their meaning.

Table 5.1- 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

5.2 Lexical Conventions

This sub clause[1] presents the lexical conventions of IDL. It defines tokens in an IDL specification and describes comments, identifiers, keywords, and literals—integer, character, and floating point constants and string literals.

An 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.

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) character, and formatting characters. Table 5.2 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 5.3.

Table 5.2- Characters

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

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

Table 5.2- Characters

Char.	Description	Char.	Description
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 5.3 lists the decimal digit characters.

Table 5.3- Decimal Digits

0 1 2 3 4 5 6 7 8 9

Table 5.4 shows the graphic characters.

Table 5.4 - Graphic Characters

Character	Description	Character	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

Table 5.4 - Graphic Characters

Character	Description	Character	Description
*	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
		³	division sign

The formatting characters are shown in Table 5.5.

Table 5.5 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

5.2.1 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.

5.2.2 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.

5.2.3 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 5.2 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 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
    };
};
```

5.2.3.1 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 that 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.

5.2.4 Keywords

The identifiers listed in Table 5.6 are reserved for use as keywords and may not be used otherwise, unless escaped with a leading underscore.

Table 5.6 - Keywords

abstract	exception	inout	provides	truncatable
any	emits	interface	public	typedef
attribute	enum	local	publishes	typeid
boolean	eventtype	long	raises	typeprefix
case	factory	module	readonly	unsigned
char	FALSE	multiple	setraises	union
component	finder	native	sequence	uses
const	fixed	Object	short	ValueBase
consumes	float	octet	string	valuetype
context	getraises	oneway	struct	void
custom	home	out	supports	wchar
default	import	primarykey	switch	wstring
double	in	private	TRUE	

Keywords must be written exactly as shown in the above list. Identifiers that collide with keywords (see 5.2.3, Identifiers) 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;
};
```

IDL specifications use the characters shown in Table 5.7 as punctuation.

Table 5.7 - Punctuation

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

In addition, the tokens listed in Table 5.8 are used by the preprocessor.

Table 5.8 - Tokens

#	##	!	||	&&

5.2.5 Literals

This sub clause describes the following literals:

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

5.2.5.1 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.

5.2.5.2 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 5.2 on page 4, Table 5.3 on page 5, and Table 5.4 on page 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 5.5 on page 7). The meaning of all other characters is implementation-dependent.

Nongraphic characters must be represented using escape sequences as defined below in Table 5.9. Note that escape sequences must be used to represent single quote and backslash characters in character literals.

Table 5.9 - 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	\'

Table 5.9 - Escape Sequences

Description	Escape Sequence
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. Because a 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. Attempts 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.

5.2.5.3 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.

5.2.5.4 String Literals

A string literal is a sequence of characters (as defined in 5.2.5.2, Character Literals), with the exception of the character with numeric value 0, 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. 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.

5.2.5.5 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.

5.3 Preprocessing

IDL is preprocessed according to the specification of the preprocessor in ISO/IEC 14882:2003. The preprocessor may be implemented as a separate process or built into the IDL compiler.

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 IDL; they may appear anywhere and have effects that last (independent of the IDL scoping rules) until the end of the translation unit. The textual location of 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 IDL token (see 5.2.1, Tokens), 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 IDL specifications. Text in files included with a #include directive is treated as if it appeared in the including file, except that **RepositoryId** related pragmas are handled in a special way. The special handling of these pragmas is described in the CORBA 3.2 specification, Sub clause 14.7, RepositoryIds.

Note that whether a particular IDL compiler generates code for included files is an implementation-specific issue. To support separate compilation, IDL compilers may not generate code for included files, or do so only if explicitly instructed.

5.4 IDL Grammar

(1) <specification>::=<import>* <definition>$^{+}$

(2) <definition>::=<type_dcl> “;”
| <const_dcl> “;”
| <except_dcl> “;”
| <interface> “;”
| <module> “;”
| <value> “;”
| <type_id_dcl> “;”
| <type_prefix_dcl> “;”
| <event> “;”
| <component> “;”
| <home_dcl> “;”

(3) <module>::=“module” <identifier> “{“ <definition>$^{+}$ “}”

(4) <interface>::=<interface_dcl>
| <forward_dcl>

(5) <interface_dcl>::=<interface_header> “{” <interface_body> “}”

(6) <forward_dcl>::=[“abstract” | “local”] “interface” <identifier>

(7) <interface_header>::=[“abstract” | “local”] “interface” <identifier>
[<interface_inheritance_spec>]

(8) <interface_body>::=<export>*

(9) <export>::=<type_dcl> “;”
| <const_dcl> “;”
| <except_dcl> “;”
| <attr_dcl> “;”
| <op_dcl> “;”
| <type_id_dcl> “;”
| <type_prefix_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>] “)”
[<raises_expr>] “;”

(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” <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>
| <constr_forward_decl>

(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_attr_spec>
| <attr_spec>

(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”

(99) <constr_forward_decl>::=“struct” <identifier>
| “union” <identifier>

(100) <import> ::= “import” <imported_scope> “;”

(101) <imported_scope> ::= <scoped_name> | <string_literal>

(102) <type_id_dcl> ::=“typeid” <scoped_name> <string_literal>

(103) <type_prefix_dcl>::=“typeprefix” <scoped_name> <string_literal>

(104) <readonly_attr_spec> ::= “readonly” “attribute” <param_type_spec>
<readonly_attr_declarator>

(105) <readonly_attr_declarator>::= <simple_declarator> <raises_expr>
| <simple_declarator>
{ “,” <simple_declarator> }*

(106) <attr_spec> ::= “attribute” <param_type_spec>
<attr_declarator>

(107) <attr_declarator> ::=<simple_declarator> <attr_raises_expr>
| <simple_declarator>
{ “,” <simple_declarator> }*

(108) <attr_raises_expr> ::=<get_excep_expr> [<set_excep_expr>]
| <set_excep_expr>

(109) <get_excep_expr> ::= “getraises” <exception_list>

(110) <set_excep_expr> ::= “setraises” <exception_list>

(111) <exception_list>::= “(” <scoped_name>
{ “,” <scoped_name> } * “)”

NOTE: Grammar rules 1 through 111 with the exception of the last three lines of rule 2 constitutes the portion of IDL that

is not related to components.

(112) <component> ::=<component_dcl>
| <component_forward_dcl>

(113) <component_forward_dcl> ::= “component” <identifier>

(114) <component_dcl> ::= <component_header>
“{” <component_body> “}”

(115) <component_header> ::= “component” <identifier>
[<component_inheritance_spec>]
[<supported_interface_spec>]

(116) <supported_interface_spec> ::= “supports” <scoped_name>
{ “,” <scoped_name> }*

(117) <component_inheritance_spec> ::= “:” <scoped_name>

(118) <component_body> ::=<component_export>*

(119) <component_export> ::=<provides_dcl> “;”
| <uses_dcl> “;”
| <emits_dcl> “;”
| <publishes_dcl> “;”
| <consumes_dcl> “;”
| <attr_dcl> “;”

(120) <provides_dcl> ::= “provides” <interface_type> <identifier>

(121) <interface_type> ::= <scoped_name>
| “Object”

(122) <uses_dcl> ::= “uses” [“multiple”]
< interface_type> <identifier>

(123) <emits_dcl> ::= “emits” <scoped_name> <identifier>

(124) <publishes_dcl> ::= “publishes” <scoped_name> <identifier>

(125) <consumes_dcl> ::= “consumes” <scoped_name> <identifier>

(126) <home_dcl> ::= <home_header> <home_body>

(127) <home_header> ::= “home” <identifier>
[<home_inheritance_spec>]
[<supported_interface_spec>]
“manages” <scoped_name>
[<primary_key_spec>]

(128) <home_inheritance_spec> ::= “:” <scoped_name>

(129) <primary_key_spec> ::= “primarykey” <scoped_name>

(130) <home_body> ::= “{” <home_export>* “}”

(131) <home_export ::= <export>
| <factory_dcl> “;”
| <finder_dcl> “;”

(132) <factory_dcl> ::= “factory” <identifier>
“(“ [<init_param_decls>] “)”
[<raises_expr>]

(133) <finder_dcl> ::= “finder” <identifier>
“(“ [<init_param_decls>] “)”
[<raises_expr>]

(134) <event> ::= (<event_dcl> | <event_abs_dcl> |
<event_forward_dcl>)

(135) <event_forward_dcl> ::=[“abstract”] “eventtype” <identifier>
(136) <event_abs_dcl> ::=“abstract” “eventtype” <identifie
[<value_inheritance_spec>]
“{” <export>* “}”
(137) <event_dcl>::=<event_header> “{” <value_element> * “}”
(138) <event_header>::=[“custom”] “eventtype”
<identifier> [<value_inheritance_spec>]

5.5 IDL Specification

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

(1)<specification>::=<import>* <definition>$^{+}$
(2) <definition>::=<type_dcl> “;”
| <const_dcl> “;”
| <except_dcl> “;”
| <interface> “;”
| <module> “;”
| <value> “;”
| <type_id_dcl> “;”
| <type_prefix_dcl> “;”
| <event> “;”
| <component> “;”
| <home_dcl> “;”

See Import Declaration on page 19, for the specification of <import>.

See Module Declaration on page 20, for the specification of <module>.

See Interface Declaration on page 21, for the specification of <interface>.

See Value Declaration on page 26, for the specification of <value>.

See Constant Declaration on page 31, Type Declaration on page 35, and Exception Declaration on page 47 respectively for specifications of **<const_dcl>**, **<type_dcl>**, and **<except_dcl>**.

See Repository Identity Related Declarations on page 51, for specification of Repository Identity declarations which include <type_id_dcl> and <type_prefix_dcl>.

See Event Declaration on page 53, for specification of <event>.

See Component Declaration on page 54, for specification of <component>.

See Section 5.18, <$paratext>, on page 59, for specification of <home_dcl>.

5.6 Import Declaration

The grammar for the import statement is described by the following Backus Naur Form (BNF):

(100)<import> ::= “import” <imported_scope> “;”
(101) <imported_scope> ::= <scoped_name> | <string_literal>

The **<imported_scope>** non-terminal may be either a fully-qualified scoped name denoting an IDL name scope, or a string containing the interface repository ID of an IDL name scope, i.e., a definition object in the repository whose interface derives from **CORBA::Container**.

The definition of import obviates the need to define the meaning of IDL constructs in terms of "file scopes." This International Standard defines the concepts of a specification as a unit of IDL expression. In the abstract, a specification consists of a finite sequence of ISO Latin-1 characters that form a legal IDL sentence. The physical representation of the specification is of no consequence to the definition of IDL, though it is generally associated with a file in practice.

Any scoped name that begins with the scope token ("**::**") is resolved relative to the global scope of the specification in which it is defined. In isolation, the scope token represents the scope of the specification in which it occurs.

A specification that imports name scopes must be interpreted in the context of a well-defined set of IDL specifications whose union constitutes the space from within which name scopes are imported. By "a well-defined set of IDL specifications," we mean any identifiable representation of IDL specifications, such as an interface repository. The specific representation from which name scopes are imported is not specified, nor is the means by which importing is implemented, nor is the means by which a particular set of IDL specifications (such as an interface repository) is associated with the context in which the importing specification is to be interpreted.

The effects of an import statement are as follows:

- The contents of the specified name scope are visible in the context of the importing specification. Names that occur in IDL declarations within the importing specification may be resolved to definitions in imported scopes.
- Imported IDL name scopes exist in the same space as names defined in subsequent declarations in the importing specification.
- IDL module definitions may re-open modules defined in imported name scopes.
- Importing an inner name scope (i.e., a name scope nested within one or more enclosing name scopes) does not implicitly import the contents of any of the enclosing name scopes.
- When a name scope is imported, the names of the enclosing scopes in the fully-qualified pathname of the enclosing scope are *exposed* within the context of the importing specification, but their contents are not imported. An importing specification may not redefine or reopen a name scope that has been exposed (but not imported) by an import statement.
- Importing a name scope recursively imports all name scopes nested within it.
- For the purposes of this International Standard, name scopes that can be imported (i.e., specified in an import statement) include the following: **modules**, **interfaces**, **valuetypes**, and **eventtypes**.
- Redundant imports (e.g., importing an inner scope and one of its enclosing scopes in the same specification) are disregarded. The union of all imported scopes is visible to the importing program.
- This International Standard does not define a particular form for generated stubs and skeletons in any given programming language. In particular, it does not imply any normative relationship between units specification and units of generation and/or compilation for any language mapping.

5.7 Module Declaration

A module definition satisfies the following syntax:

(3)<module>::="module" <identifier> "{" <definition>$^{+}$ "}"

The module construct is used to scope IDL identifiers; see CORBA Module on page 68 for details.

5.8 Interface Declaration

An interface definition satisfies the following syntax:

(4) <interface>::=<interface_dcl>
| <forward_dcl>
(5) <interface_dcl>::=<interface_header> “{” <interface_body> “}”
(6) <forward_dcl>::=[“abstract” | “local”] “interface” <identifier>
(7) <interface_header>::=[“abstract” | “local”] “interface” <identifier>
[<interface_inheritance_spec>]
(8) <interface_body>::=<export>*
(9) <export>::=<type_dcl> “;”
| <const_dcl> “;”
| <except_dcl> “;”
| <attr_dcl> “;”
| <op_dcl> “;”
| <type_id_decl> “;”
| <type_prefix_decl> “;”

5.8.1 Interface Header

The interface header consists of three elements:

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

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 sub clauses. 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 Abstract Interface on page 25. They also follow different language mapping rules.

Local interfaces have slightly different rules and semantics from “regular” interfaces, as described in Local Interface on page 25. They also follow different language mapping rules.

5.8.2 Interface Inheritance Specification

The syntax for inheritance is as follows:

(10) <interface_inheritance_spec>::=“:” <interface_name>
{ “,” <interface_name> }*

(11) <interface_name>::=<scoped_name>

(12) <scoped_name>::=<identifier>

| "::" <identifier>

| <scoped_name> "::" <identifier>

Each **<scoped_name>** in an **<interface_inheritance_spec>** must be the name of a previously defined interface or an alias to a previously defined interface. See Interface Inheritance on page 23 for the description of inheritance.

5.8.3 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 Constant Declaration on page 31.
- Type declarations, which specify the type definitions that the interface exports. Type declaration syntax is described in Type Declaration on page 35.
- Exception declarations, which specify the exception structures that the interface exports. Exception declaration syntax is described in Exception Declaration on page 47.
- Attribute declarations, which specify the associated attributes exported by the interface. Attribute declaration syntax is described in Attribute Declaration on page 50.
- 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 that may be returned as a result of an invocation, and contextual information that may affect method dispatch. Operation declaration syntax is described in Operation Declaration on page 47.

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

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

5.8.4 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 is: optionally either the keyword **abstract** or the keyword **local**, followed by the keyword **interface**, followed by an <identifier> that names the interface.

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
};
```

5.8.5 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 that have been inherited; the scope rules for such names are described in Names and Scoping on page 69.

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 7.1. This “diamond” shape is legal, as is the definition of E on the right.

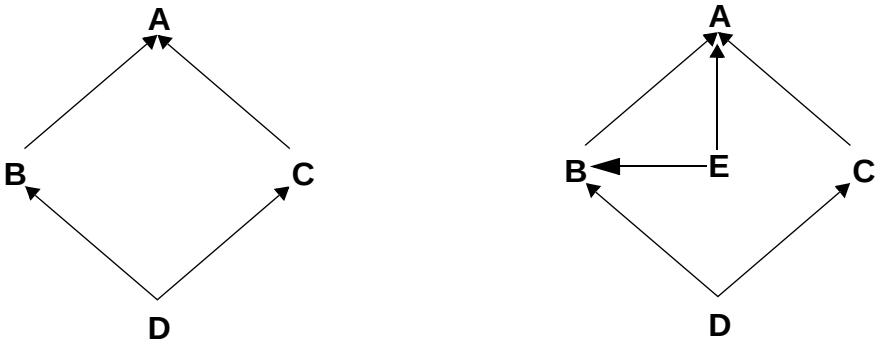

Figure 7.1 - Legal Multiple Inheritance Example

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 defined in base interfaces, both direct and indirect, to be visible in 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 and 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
};
```

For a complete summary of allowable inheritance and supporting relationships among interfaces and valuetypes see Table 5.6 on page 9.

5.8.6 Abstract Interface

An interface declaration containing the keyword **abstract** in its header, declares an abstract interface. The following special rules apply to abstract interfaces:

- Abstract interfaces may only inherit from other abstract interfaces.
- Value types may support any number of abstract interfaces.

See the CORBA 3.2 specification, Sub clause 10.2 “Semantics of Abstract Interfaces” for CORBA implementation semantics associated with abstract interfaces.

For a complete summary of allowable inheritance and supporting relationships among interfaces and valuetypes see Table 5.10 on page 30.

5.8.7 Local Interface

An interface declaration containing the keyword **local** in its header declares a local interface. An interface declaration not containing the keyword **local** is referred to as an unconstrained interface. An object implementing a local interface is referred to as a local object. The following special rules apply to local interfaces:

- A local interface may inherit from other local or unconstrained interfaces.
- An unconstrained interface may not inherit from a local interface. An interface derived from a local interface must be explicitly declared local.
- A valuetype may support a local interface.

- Any IDL type, including an unconstrained interface, may appear as a parameter, attribute, return type, or exception declaration of a local interface.
- A local interface is a local type, as is any non-interface type declaration constructed using a local interface or other local type. For example, a struct, union, or exception with a member that is a local interface is also itself a local type.
- A local type may be used as a parameter, attribute, return type, or exception declaration of a local interface or of a valuetype.
- A local type may not appear as a parameter, attribute, return type, or exception declaration of an unconstrained interface.

For a complete summary of allowable inheritance and supporting relationships among interfaces and valuetypes see Table 5.10 on page 30.

See the CORBA 3.2 specification, Sub clause 8.3.14 “Local Object Operations” for CORBA implementation semantics associated with local objects.

5.9 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:

(13) <value> ::= (<value_dcl> | <value_abs_dcl> | <value_box_dcl> | <value_forward_dcl>)

5.9.1 Regular Value Type

A regular value type satisfies the following syntax:

(17)<value_dcl> ::=<value_header> “{“ < value_element>* “}”
(18) <value_header> ::=[“custom”] “valuetype” <identifier>
[<value_inheritance_spec>]
(21)<value_element> ::=<export>
| < state_member> |
| <init_dcl>

5.9.1.1 Value Header

The value header consists of two elements:

1. The value type’s name and optional modifier specifying whether the value type uses custom marshaling.
2. An optional value inheritance specification. The value inheritance specification is described below.

5.9.1.2 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.

5.9.1.3 Value Inheritance Specification

(19)<value_inheritance_spec> ::=[":" ["truncatable"] <value_name>
{ "," <value_name> }*]
["supports" <interface_name>
{ "," <interface_name> }*]

(20) <value_name> ::=<scoped_name>

Each **<value_name>** in a **<value_inheritance_spec>** must be the name of a previously defined value type or an alias to a previously defined value type. Each **<interface_name>** in a **<value_inheritance_spec>** must be the name of a previously defined interface or an alias to a previously defined interface. See "Valuetype Inheritance" for the description of value type inheritance.

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

A valuetype that supports a local interface does not itself become *local* (i.e., unmarshalable) as a result of that support.

5.9.1.4 State Members

(22)<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.

A valuetype that has a state member that is *local* (i.e., non-marshalable like a local interface), is itself rendered *local*. That is, such valuetypes behave similar to local interfaces when an attempt is made to marshal them.

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.

5.9.1.5 Initializers

(23) <init_dcl> ::="factory" <identifier>
"(" [<init_param_decls>] ")"
[<raises_expr>] ";"

(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"

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. Initializers defined in a valuetype are not inherited by derived valuetypes, and hence the names of the initializers are free to be reused in a derived valuetype.

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.

5.9.1.6 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 {};
```

5.9.2 Boxed Value Type

(15)<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."

Since a value box of a valuetype adds no additional properties to a valuetype, it is an error to box valuetypes.

Value box 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 IDs) 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.

Any IDL type may be used to declare a value box except for a valuetype.

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.

5.9.3 Abstract Value Type

(16) <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.

5.9.4 Value Forward Declaration

(14)<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.

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.

It is illegal for a value type to support a forward-declared interface whose definition has not yet been seen.

5.9.5 Valuetype Inheritance

The terminology that is used to describe value type inheritance is directly analogous to that used to describe interface inheritance (see Interface Inheritance on page 23).

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 Interface Inheritance on page 23 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.

While a valuetype may only directly support one interface, it is possible for the valuetype to support other interfaces as well through inheritance. In this case, the supported interface must be derived, directly or indirectly, from each interface that the valuetype supports through inheritance. This rule does not apply to abstract interfaces that the valuetype supports. For example:

```
interface I1 { };
interface I2 { };
interface I3: I1, I2 { };

abstract valuetype V1 supports I1 { };
abstract valuetype V2 supports I2 { };
valuetype V3: V1, V2 supports I3 { }; // legal
valuetype V4: V1 supports I2 { }; // illegal
```

A stateful value that derives from another stateful value may specify that it is **truncatable**. This means that it is to "truncate" (see the CORBA 3.2 specification, Sub clause 9.2.5.3 "Value instance -> Value type") 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 5.10

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 single	supports multiple	multiple	no	no

Table 5.10

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

5.10 Constant Declaration

This sub clause describes the syntax for constant declarations.

5.10.1 Syntax

The syntax for a constant declaration is:

```
(27)<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>

5.10.2 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.

Octet literals have integer value in the range 0..255. If the right hand side of an **octet** constant declaration is outside this range it shall be flagged as a compile time error.

Integer literals have positive integer values. Constant integer literals are considered **unsigned long** unless the value is too large, then they are considered **unsigned long long**. Unary minus is considered an operator, not a part of an integer literal. Only integer values can be assigned to integer type (**short**, **long**, **long long**) constants, and **octet** constants. Only positive integer values can be assigned to unsigned integer type constants. If the value of the right hand side of an integer constant declaration is too large to fit in the actual type of the constant on the left hand side, for example

```
const short s = 655592;
```

or is inappropriate for the actual type of the left hand side, for example

```
const octet o = -54;
```

it shall be flagged as a compile time error.

Floating point literals have floating point values. Only floating point values can be assigned to floating point type (**float**, **double**, **long double**) constants. Constant floating point literals are considered **double** unless the value is too large, then they are considered **long double**. If the value of the right hand side is too large to fit in the actual type of the constant to which it is being assigned, it shall be flagged as a compile time error. Truncation on the right for floating point types is OK.

Fixed point literals have fixed point values. Only fixed point values can be assigned to fixed point type constants. If the fixed point value in the expression on the right hand side is too large to fit in the actual fixed point type of the constant on the left hand side, then it shall be flagged as a compile time error. Truncation on the right for fixed point types is OK.

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**.

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

Integer expressions are evaluated using the imputed type of each argument of a binary operator in turn. If either argument is **unsigned long long**, use **unsigned long long**. If either argument is **long long**, use **long long**. If either argument is **unsigned long**., use **unsigned long**. Otherwise use **long**. The final result of an integer arithmetic expression must fit in the range of the declared type of the constant, otherwise an error shall be flagged by the compiler. In addition to the integer types, the final result of an integer arithmetic expression can be assigned to an **octet** constant, subject to it fitting in the range for **octet** type.

Floating point expressions are evaluated using the imputed type of each argument of a binary operator in turn. If either argument is **long double**, use **long double**. Otherwise use **double**. The final result of a floating point arithmetic expression must fit in the range of the declared type of the constant, otherwise an error shall be flagged by the compiler.

Fixed-point decimal constant expressions are evaluated as follows. A fixed-point literal has the apparent number of total and fractional digits. For example, **0123.450d** is considered to be **fixed<7,3>** and **3000.00d** is **fixed<6,2>**. 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) + sinf, sinf>

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 non-negative, then the remainder is non-negative; 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.

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 5.21, Names and Scoping. 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
```

5.11 Type Declaration

IDL provides constructs for naming data types; that is, it provides C language-like declarations that associate an identifier with a type. 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:

```
(42) <type_dcl>::="typedef" <type_declarator>
                          | <struct_type>
                          | <union_type>
                          | <enum_type>
                          | "native" <simple_declarator>
                          | <constr_forward_decl>
(43) <type_declarator>::=<type_spec> <declarators>
```

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

```
(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>

The **<scoped_name>** in **<simple_type_spec>** must be a previously defined type introduced by an interface declaration (**<interface_dcl>** - see 5.8, Interface Declaration), a value declaration (**<value_dcl>**, **<value_box_dcl>** or **<abstract_value_dcl>** - see 5.9, Value Declaration) or a type declaration (**<type_dcl>** - see 5.11, Type Declaration). Note that exceptions are not considered types in this context.

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

5.11.1 Basic Types

The syntax for the supported basic types is as follows:

(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”

Each IDL data type is mapped to a native data type via the appropriate language mapping. Conversion errors between 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 system exceptions that are to be raised in such situations are defined in the CORBA 3.2 specification, Sub clause 8.12 “Exceptions.”

5.11.1.1 Integer Types

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 5.11.

Table 5.11

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	$0 .. 2^{64} - 1$

5.11.1.2 Floating-Point Types

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.

5.11.1.3 Char Type

IDL defines a **char** data type that is an 8-bit quantity that (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 IDL (i.e., the space, alphabetic, digit, and graphic characters defined in Table 5.2 on page 4, Table 5.3 on page 5, and Table 5.4 on page 5). The meaning and representation of the null and formatting characters (see Table 5.5 on page 7) 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.

5.11.1.4 Wide Char Type

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

5.11.1.5 Boolean Type

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

5.11.1.6 Octet Type

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

5.11.1.7 Any Type

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

An **any** logically contains a **TypeCode** (see the CORBA 3.2 specification, Sub clause 8.11 “TypeCodes”) 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.

5.11.2 Constructed Types

Structs, **unions**, and **enums** are the constructed types. Their syntax is presented below:

```
(42)<type_dcl>::=“typedef” <type_declarator>
                        |  <struct_type>
                        |  <union_type>
                        |  <enum_type>
                        |  “native” <simple_declarator>
                        |  <constr_forward_decl>
(48)<constr_type_spec>::=<struct_type>
                        |  <union_type>
                        |  <enum_type>
(99)<constr_forward_decl>::=“struct” <identifier>
                        |  “union” <identifier>
```

5.11.2.1 Structures

The syntax for **struct** type is:

```
(69)<struct_type>::=“struct” <identifier> “{” <member_list> “}”
(70)  <member_list>::=<member>+
(71)  <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.

5.11.2.2 Discriminated Unions

The discriminated **union** syntax is:

```
(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>

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. 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.

The values of the constant expressions for the case labels of a single union definition must be distinct. A union type can contain a default label only where the values given in the non-default labels do not cover the entire range of the union's discriminant type.

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

NOTE: While any ISO Latin-1 (8859.1) IDL character literal may be used in a **<case_label>** in a union definition whose discriminator type is **char**, not all of these characters are present in all transmission code sets that may be negotiated by GIOP or in all native code sets that may be used by implementation language compilers and runtimes. When an attempt is made to marshal to CDR a **union** whose discriminator value of **char** type is not available in the negotiated transmission code set, or to demarshal from CDR a **union** whose discriminator value of **char** type is not available in the native code set, a **DATA_CONVERSION** system exception is raised. Therefore, to ensure portability and interoperability, care must be exercised when assigning the **<case_label>** for a **union** member whose discriminator type is **char**. Due to these issues, use of **char** types as the discriminator type for **union**s is not recommended.

5.11.2.3 Constructed Recursive Types and IForward Declarations

The IDL syntax allows the generation of recursive structures and unions via members that have a sequence type. The element type of a recursive sequence struct or union member must identify a struct, union, or valuetype. (A valuetype is allowed to have a member of its own type either directly or indirectly through a member of a constructed type—see 5.9.1.6, Value Type Example.) For example, the following is legal:

```
struct Foo {
    long value;
    sequence<Foo> chain;      // Deprecated (see  Section 5.11.6)
}
```

See Sequences on page 42 for details of the **sequence** template type.

IDL supports recursive types via a forward declaration for structures and unions (as well as for valuetypes—see 5.9.1.6, Value Type Example). Because anonymous types are deprecated (see Deprecated Anonymous Types on page 44), the previous example is better written as:

```
struct Foo;                              // Forward declaration
typedef sequence<Foo> FooSeq;
struct Foo {
    long value;
    FooSeq chain;
};
```

The forward declaration for the structure enables the definition of the sequence type **FooSeq**, which is used as the type of the recursive member.

Forward declarations are legal for structures and unions. A structure or union type is termed incomplete until its full definition is provided; that is, until the scope of the structure or union definition is closed by a terminating "}." For example:

```
struct Foo;      // Introduces Foo type name,
                 // Foo is incomplete now
                 // ...
struct Foo {
    // ...
};               // Foo is complete at this point
```

If a structure or union is forward declared, a definition of that structure or union must follow the forward declaration in the same source file. Compilers shall issue a diagnostic if this rule is violated. Multiple forward declarations of the same structure or union are legal.

If a sequence member of a structure or union refers to an incomplete type, the structure or union itself remains incomplete until the member's definition is completed. For example:

```
struct Foo;
typedef sequence<Foo> FooSeq;
struct Bar {
    long value;
    FooSeq chain; //Use of incomplete type
};                //Bar itself remains incomplete
struct Foo {
    // ...
};                //Foo and Bar are complete
```

Compilers shall issue a diagnostic if this rule is violated.

Recursive definitions can span multiple levels. For example:

```
union Bar;                  // Forward declaration
typedef sequence<Bar> BarSeq;
union Bar switch(long) {    // Define incomplete union
    case 0:
        long l_mem;
    case 1:
        struct Foo {
            double d_mem;
            BarSeq nested;      // OK, recurse on enclosing
                                // incomplete type
    } s_mem;
};
```

An incomplete type can only appear as the element type of a sequence definition. A sequence with incomplete element type is termed an *incomplete sequence type*. For example:

```
struct Foo;                         // Forward declaration
typedef sequence<Foo> FooSeq;       // incomplete
```

An incomplete sequence type can appear only as the element type of another sequence, or as the member type of a structure or union definition. For example:

```
struct Foo;                         // Forward declaration
typedef sequence<Foo> FooSeq;       // OK
typedef sequence<FooSeq> FooTree; // OK

interface I {
    FooSeq op1();       // Illegal, FooSeq is incomplete
    void op2(           // Illegal, FooTree is incomplete
        in FooTree t
    );
};

struct Foo {            // Provide definition of Foo
    long l_mem;
    FooSeq chain;       // OK
    FooTree tree;       // OK
};

interface J {
    FooSeq op1();       // OK, FooSeq is complete
    void op2(
        in FooTree t    // OK, FooTree is complete
    );
};
```

Compilers shall issue a diagnostic if this rule is violated.

5.11.2.4 Enumerations

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

(78)<enum_type>::=“enum” <identifier>
“{” <enumerator> { “,” <enumerator> }* “}”
(79) <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 that 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.

5.11.3 Template Types

The template types are:

(47)<template_type_spec>::=<sequence_type>
| <string_type>
| <wide_string_type>
| <fixed_pt_type>

5.11.3.1 Sequences

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:

(80)<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. If no maximum size is specified, size of the sequence is unspecified (unbounded).

Prior to passing a bounded or unbounded 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.

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.

5.11.3.2 Strings

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:

(81)<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.

5.11.3.3 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:

(82) <wide_string_type>::=“wstring” “<” <positive_int_const> “>”
| “wstring”

5.11.3.4 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. The syntax of fixed type is:

(96)<fixed_pt_type>::=“fixed” “<“ <positive_int_const> “,” <positive_int_const> “>”
(97) <fixed_pt_const_type>::=“fixed”

5.11.4 Complex Declarator

5.11.4.1 Arrays

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

The syntax for arrays is:

(83)<array_declarator>::=<identifier> <fixed_array_size>$^{+}$
(84) <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.

5.11.5 Native Types

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:

(42)<type_dcl>::=“native” <simple_declarator>
(51)<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 only to define operation parameters, results, and exceptions. If a native type is used for an exception, it must be mapped to a type in a programming language that can be used as an exception. Native type parameters are permitted only in operations of **local interface**s or **valuetype**s. Any attempt to transmit a value of a native type in a remote invocation may raise the MARSHAL standard system 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 IDL language or to the IDL compiler.

5.11.6 Deprecated Anonymous Types

IDL currently permits the use of anonymous types in a number of places. For example:

```
struct Foo {
    long value;
    sequence<Foo> chain;      // Legal (but deprecated)
}
```

Anonymous types cause a number of problems for language mappings and are therefore deprecated by this International Standard. Anonymous types will be removed in a future version, so new IDL should avoid use of anonymous types and use a typedef to name such types instead. Compilers need not issue a warning if a deprecated construct is encountered.

The following (non-exhaustive) examples illustrate deprecated uses of anonymous types.

Anonymous bounded string and bounded wide string types are deprecated. This rule affects constant definitions, attribute declarations, return value and parameter type declarations, sequence and array element declarations, and structure, union, exception, and valuetype member declarations. For example:

```
const string<5> GREETING = “Hello”;          // Deprecated

interface Foo {
    readonly attribute wstring<5> name;      // Deprecated
    wstring<5> op(in wstring<5> param);      // Deprecated
};
typedef sequence<wstring<5> > WS5Seq;        // Deprecated
typedef wstring<5> NameVector [10];          // Deprecated
struct A {
    wstring<5> mem;                          // Deprecated
};
// Anonymous member type in unions, exceptions,
// and valuetypes are deprecated as well.
```

This is better written as:

```
typedef string<5> GreetingType;
const GreetingType GREETING = “Hello”;

typedef wstring<5> ShortWName;
interface Foo {
    readonly attribute ShortWName name;
    ShortWName op(in ShortWName param);
};
typedef sequence<ShortWName> NameSeq;
typedef ShortWName NameVector[10];
struct A {
    GreetingType mem;
};
```

Anonymous fixed-point types are deprecated. This rule affects attribute declarations, return value and parameter type declarations, sequence and array element declarations, and structure, union, exception, and valuetype member declarations.

```
struct Foo {
    fixed<10,5> member;                      // Deprecated
};
```

This is better written as:

```
typedef fixed<10,5> MyType;
struct Foo {
    MyType member;
};
```

Anonymous member types in structures, unions, exceptions, and valuetypes are deprecated:

```
union U switch(long) {
    case 1:
        long array_mem[10];                    // Deprecated
    case 2:
        sequence<long> seq_mem;                // Deprecated
    case 3:
        string<5> bstring_mem;
};
```

This is better written as:

```
typedef long LongArray[10];
typedef sequence<long> LongSeq;
typedef string<5> ShortName;
union U switch (long) {
    case 1:
        LongArray array_mem;
    case 2:
        LongSeq seq_mem;
    case 3:
        ShortName bstring_mem;
};
```

Anonymous array and sequence elements are deprecated:

```
typedef sequence<sequence<long> > NumberTree;   // Deprecated
typedef fixed<10,2> FixedArray[10];
```

This is better written as:

```
typedef sequence<long> ListOfNumbers;
typedef sequence<ListOfNumbers> NumberTree;
typedef fixed<10,2> Fixed_10_2;
typedef Fixed_10_2 FixedArray[10];
```

The preceding examples are not exhaustive. They simply illustrate the rule that, for a type to be used in the definition of another type, constant, attribute, return value, parameter, or member, that type must have a name. Note that the following example is not deprecated (even though stylistically poor):

```
struct Foo {
    struct Bar {
        long l_mem;
        double d_mem;
    } bar_mem_1;            // OK, not anonymous
    Bar bar_mem_2;          // OK, not anonymous
};
typedef sequence<Foo::Bar> FooBarSeq;      // Scoped names are OK
```

5.12 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:

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

Each exception is characterized by its 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.

An identifier declared to be an exception identifier may thereafter appear only in a raises clause of an operation declaration, and nowhere else.

A set of standard system exceptions is defined corresponding to standard run-time errors, which may occur during the execution of a request. These standard system exceptions are documented in the CORBA 3.2 specification, Sub clause 8.12 “Exceptions.”

5.13 Operation Declaration

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

(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”

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 5.13.1, Operation Attribute.
- The type of the operation’s return result; the type may be any type that can be defined in 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 5.13.2, Parameter Declarations.
- An optional raises expression that indicates which exceptions may be raised as a result of an invocation of this operation. Raises expressions are described in 5.13.3, Raises Expressions.
- An optional context expression that indicates which elements of the request context may be consulted by the method that implements the operation. Context expressions are described in 5.13.4, Context Expressions.

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

5.13.1 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:

(88)<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 system 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.

5.13.2 Parameter Declarations

Parameter declarations in IDL operation declarations have the following syntax:

(90)<parameter_dcls>::=“(” <param_dcl> { “,” <param_dcl> }* “)”
| “(” “)”
(91) <param_dcl>::=<param_attribute> <param_type_spec> <simple_declarator>
(92) <param_attribute>::=“in”
| “out”
| “inout”
(95)<param_type_spec>::=<base_type_spec>
| <string_type>
| <wide_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.

5.13.3 Raises Expressions

There are two kinds of raises expressions as described in this sub clause.

5.13.3.1 Raises Expression

A **raises** expression specifies which exceptions may be raised as a result of an invocation of the operation or accessing (invoking the _get operation of) a readonly attribute. The syntax for its specification is as follows:

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

The **<scoped_name>**s in the **raises** expression must be previously defined exceptions or native types. If a native type is used as an exception for an operation, the operation must appear in either a local interface or a valuetype.

In addition to any operation-specific exceptions specified in the **raises** expression, there are a standard set of system exceptions that may be signalled by the ORB. These standard system exceptions are described in the CORBA 3.2 specification, Sub clause 8.12.3 “Standard System Exception Definitions.” However, standard system 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 system exceptions.

5.13.3.2 getraises and setraises Expressions

getraises and **setraises** expressions specify which exceptions may be raised as a result of an invocation of the accessor (**_get**) and a mutator (**_set**) functions of an attribute. The syntax for its specification is as follows:

(108)<attr_raises_expr> ::=<get_excep_expr> [<set_excep_expr>]
| <set_excep_expr>
(109) <get_excep_expr> ::= “getraises” <exception_list>
(110) <set_excep_expr> ::= “setraises” <exception_list>
(111) <exception_list::= “(” <scoped_name>
{ “,” <scoped_name> } * “)”

The **<scoped_name>**s in the **getraises** and **setraises** expressions must be previously defined exceptions.

In addition to any attribute-specific exceptions specified in the **getraises** and **setraises** expressions, there are a standard set of exceptions that may be signalled by the ORB. These standard exceptions are described in 8.12.3, Standard System Exception Definitions. However, standard exceptions may *not* be listed in a **getraises** or **setraises** expression.

The absence of a **getraises** or **setraises** expression on an attribute implies that there are no accessor-specific or mutator-exceptions respectively. Invocations of such an accessor or mutator are still liable to receive one of the standard exceptions.

NOTE: The exceptions associated with the accessor operation corresponding to a **readonly attribute** is specified using a simple **raises** expression as specified in 5.13.3.1, Raises Expression. The **getraises** and **setraises** expressions are used only in **attribute**s that are not **readonly**.

5.13.4 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:

(94)<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 a non-empty string. If the character '*' appears in **string_literal**, it must appear only once, as the last character of **string_literal**, and must be preceded by one or more characters other than '*'.

The mechanism by which a client associates values with the context identifiers is described in the CORBA 3.2 specification, Sub clause 8.6 "Context Object."

5.14 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:

```
(85)<attr_dcl>::= <readonly_attr_spec>
                              |   <attr_spec>
(104)<readonly_attr_spec> ::= "readonly" "attribute" <param_type_spec>  <readonly_attr_declarator>
(105)  <readonly_attr_declarator>::=  <simple_declarator> <raises_expr>
                              |  <simple_declarator>
                                 { "," <simple_declarator> }*
(106)  <attr_spec> ::= "attribute" <param_type_spec> <attr_declarator>
(107)  <attr_declarator> ::=<simple_declarator> <attr_raises_expr>
                              |  <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 Escaped Identifiers on page 8.

```
• • •
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 IDL's name scoping rules; the accessor function names are guaranteed *not* to collide with any legal operation names specifiable in IDL.

Attributes are inherited. An attribute name *cannot* be redefined to be a different type. See 5.20, CORBA Module for more information on redefinition constraints and the handling of ambiguity.

5.15 Repository Identity Related Declarations

Two constructs that are provided for specifying information related to Repository Id are described in this sub clause.

5.15.1 Repository Identity Declaration

The syntax of a repository identity declaration is as follows:

(102)<type_id_dcl> ::="typeid" <scoped_name> <string_literal>

A repository identifier declaration includes the following elements:

- the keyword **typeid.**
- a <*scoped_name*> that denotes the named IDL construct to which the repository identifier is assigned.
- a string literal that must contain a valid repository identifier value.

The <*scoped_name*> is resolved according to normal IDL name resolution rules, based on the scope in which the declaration occurs. It must denote a previously-declared name of one of the following IDL constructs:

- module
- interface
- component
- home
- facet
- receptacle
- event sink
- event source
- finder
- factory
- event type
- value type
- value type member
- value box

- constant
- typedef
- exception
- attribute
- operation
- enum
- local

The value of the string literal is assigned as the repository identity of the specified type definition. This value will be returned as the **RepositoryId** by the interface repository definition object corresponding to the specified type definition. Language mappings constructs, such as Java helper classes, that return repository identifiers shall return the values declared for their corresponding definitions.

At most one repository identity declaration may occur for any named type definition. An attempt to redefine the repository identity for a type definition is illegal, regardless of the value of the redefinition.

If no explicit repository identity declaration exists for a type definition, the repository identifier for the type definition shall be an IDL format repository identifier, as defined in the CORBA 3.2 specification, Sub clause 14.7.1 “OMG IDL Format.”

5.15.2 Repository Identifier Prefix Declaration

The syntax of a repository identifier prefix declaration is as follows:

(103) <type_prefix_dcl>::=“typeprefix” <scoped_name> <string_literal>

A repository identifier declaration includes the following elements:

- The keyword **typeprefix**.
- A <*scoped_name*> that denotes an IDL name scope to which the prefix applies.
- A string literal that must contain the string to be prefixed to repository identifiers in the specified name scope.

The <*scoped_name*> is resolved according to normal IDL name resolution rules, based on the scope in which the declaration occurs. It must denote a previously-declared name of one of the following IDL constructs:

- module
- interface (including abstract or local interface)
- value type (including abstract, custom, and box value types)
- event type (including abstract and custom value types)
- specification scope (**::**)

The specified string is prefixed to the body of all repository identifiers in the specified name scope, whose values are assigned by default. The specified string shall be a list of one or more identifiers, separated by the “/” characters. These identifiers are arbitrarily long sequences of alphabetic, digit, underscore (“_”), hyphen (“-”), and period (“.”) characters. The string shall not contain a trailing slash (“/”), and it shall not begin with the characters underscore (“_”), hyphen (“-”) or period (“.”). To elaborate:

By “prefixed to the body of a repository identifier,” we mean that the specified string is inserted into the default IDL format repository identifier immediately after the format name and colon (“IDL:”) at the beginning of the identifier. A forward slash (‘/’) character is inserted between the end of the specified string and the remaining body of the repository identifier.

The prefix is only applied to repository identifiers whose values are not explicitly assigned by a typeid declaration. The prefix is applied to all such repository identifiers in the specified name scope, including the identifier of the construct that constitutes the name scope.

5.15.3 Repository Id Conflict

In IDL that contains both pragma prefix/ID declarations (as defined in the CORBA 3.2 specification, Sub clause 14.7.5 “Pragma Directives for RepositoryId”) and typeprefix/typeid declarations (as defined in Repository Identity Declaration on page 51 and Repository Identifier Prefix Declaration on page 52), if the repository id for an IDL element computed by using pragmas and typeid/typeprefix are not identical, it is an error. Note that this rule applies only when the repository id value computation uses explicitly declared values from declarations of both kinds. If the repository id computed using explicitly declared values of one kind conflicts with one computed with implicit values of the other kind, the repository id based on explicitly declared values shall prevail.

5.16 Event Declaration

Event type is a specialization of value type dedicated to asynchronous component communication. There are several kinds of event type declarations: “regular” event types, abstract event types, and forward declarations.

An event declaration satisfies the following syntax:

(134)<event> ::= (<event_dcl> | <event_abs_dcl> | <event_forward_dcl>)

5.16.1 Regular Event Type

A regular event type satisfies the following syntax:

(137) <event_dcl>::=<event_header> “{” <value_element> * “}”
(138) <event_header>::=[“custom”] “eventtype”
<identifier> [<value_inheritance_spec>]

5.16.1.1 Event Header

The event header consists of two elements:

- The event type’s name and optional modifier specifying whether the event type uses custom marshaling.
- An optional value inheritance specification described in 5.9.1.3, Value Inheritance Specification.

5.16.1.2 Event Element

An event can contain all the elements that a value can as described in 5.9.1.2, Value Element (i.e., attributes, operations, initializers, state members).

5.16.2 Abstract Event Type

(136)<event_abs_dcl> ::=“abstract” “eventtype” <identifier>
[<value_inheritance_spec>]
“{” <export>* “}”

Event types may also be abstract. They are called abstract because an abstract event 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 event type with an empty state is not an abstract event type.

5.16.3 Event Forward Declaration

(135)<event_forward_dcl> ::=[“abstract”] “eventtype” <identifier>

A forward declaration declares the name of an event type without defining it. This permits the definition of event types that refer to each other. The syntax consists simply of the keyword **eventtype** followed by an **<identifier>** that names the event type.

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

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

5.16.4 Eventtype Inheritance

As event type is a specialization of value type then event type inheritance is directly analogous to value inheritance (see 5.9.1.3, Value Inheritance Specification for a detailed description of the analogous properties for valuetypes). In addition, an event type could inherit from a single immediate base concrete event type, which must be the first element specified in the inheritance list of the event declaration’s IDL. It may be followed by other abstract values or events from which it inherits.

5.17 Component Declaration

5.17.1 Component

A component declaration describes an interface for a component. The salient characteristics of a component declaration are as follows:

- A component declaration specifies the name of the component.
- A component declaration may specify a list of interfaces that the component supports.
- Component declarations support single inheritance from other component definitions.
- Component declarations may include in its body any attribute declarations that are legal in normal interface declarations, together with declarations of facets and receptacles of the component, and the event sources and sinks that the component defines.

5.17.1.1 Syntax

The syntax for declaring a component is as follows:

```
(112)<component> ::=<component_dcl>
                                  |  <component_forward_dcl>
(113)  <component_forward_dcl> ::= “component” <identifier>
(114)  <component_dcl> ::= <component_header>
                “{” <component_body> “}”
```

<component_forward_dcl> is described in 5.17.1.2, Forward Declaration.

<component_header> is described in 5.17.2, Component Header.

<component_body> is described in 5.17.3, Component Body.

5.17.1.2 Forward Declaration

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

Multiple forward declarations of the same component name are legal.

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

5.17.2 Component Header

A **<component_header>** declares the primary characteristics of a component interface.

5.17.2.1 Syntax

The syntax for declaring a component header is as follows:

```
(115)  <component_header> ::= “component” <identifier>
                [ <component_inheritance_spec> ]
                [ <supported_interface_spec> ]
(116)  <supported_interface_spec> ::= “supports” <scoped_name>
                { “,” <scoped_name> }*
(117)  <component_inheritance_spec> ::= “:” <scoped_name>
```

A component header comprises the following elements:

- The keyword **component**.
- An <*identifier*> that names the component type.
- An optional <*inheritance_spec*>, consisting of a colon and a single <*scoped_name*> that must denote a previously-defined component type.
- An optional <*supported_interface_spec*> that must denote one or more previously-defined IDL interfaces.

5.17.2.2 Supported interfaces

A component may optionally support one or more interfaces. When a component definition header includes a supports clause as follows:

component <*component_name*> supports <*interface_name*> { … };

For further details see the *CORBA Components* specification, Clause 1, Supported Interfaces.

5.17.2.3 Component Inheritance

A component may optionally inherit from a component that supports one or more interfaces. This is specified by using the inheritance construct that looks like:

component <component_name> : <component_name> { ... };

The following rules apply to component inheritance:

- A derived component type may not directly support an interface.
- The interface for a derived component type is derived from the interface of its base component type.
- A component type may have at most one base component type.
- The features of a component that are inherited by the derived component are:
  - the **provides** statements
  - the **uses** statements
  - the **emits** statements
  - the **publishes** statements
  - the **consumes** statements
  - attributes

See 5.17.2.3, Component Inheritance for details of component inheritance.

5.17.3 Component Body

(118) <component_body> ::=<component_export>*
(119) <component_export> ::=<provides_dcl> “;”
| <uses_dcl> “;”
| <emits_dcl> “;”
| <publishes_dcl> “;”
| <consumes_dcl> “;”
| <attr_dcl> “;”

A component forms a naming scope, nested within the scope in which the component is declared. A component body can contain the following kinds of declarations:

- Facet declarations (**provides**)
- Receptacle declarations (**uses**)
- Event source declarations (**emits** or **publishes**)

- Event sink declarations (**consumes**)
- Attribute declarations (**attribute** and **readonly attribute**)

These declarations and their meanings are described in detail in the *CORBA Components* specification, Component Model clause, “Facets and Navigation” through “Events” sub clauses.

5.17.3.1 Facets and Navigation

A component type may provide several independent interfaces to its clients in the form of facets. Facets are intended to be the primary vehicle through which a component exposes its functional application behavior to clients during normal execution. A component may exhibit zero or more facets.

Syntax

A facet is declared with the following syntax:

```
(120)  <provides_dcl> ::= “provides” <interface_type> <identifier>
(121)  <interface_type> ::= <scoped_name>
                          |  “Object”
```

The interface type shall be either the keyword **Object**, or a scoped name that denotes a previously-declared interface type that is not a component interface (i.e., is not the interface corresponding to a component definition). The identifier names the facet within the scope of the component, allowing multiple facets of the same type to be provided by the component.

See the *CORBA Components* specification, Component Model clause, “Facets and Navigation” for further details.

5.17.3.2 Receptacles

A component definition can describe the ability to accept object references upon which the component may invoke operations. When a component accepts an object reference in this manner, the relationship between the component and the referent object is called a *connection*; they are said to be *connected*. The conceptual point of connection is called a *receptacle*. A receptacle is an abstraction that is concretely manifested on a component as a set of operations for establishing and managing connections. A component may exhibit zero or more receptacles.

Syntax

The syntax for describing a receptacle is as follows:

```
(122)  <uses_dcl> ::= “uses” [ “multiple” ]
                    < interface_type> <identifier>
```

A receptacle declaration comprises the following elements:

- The keyword **uses**.
- The optional keyword **multiple**. The presence of this keyword indicates that the receptacle may accept multiple connections simultaneously, and results in different operations on the component’s associated interface.
- An <*interface_type*>, which must be either the keyword **Object** or a scoped name that denotes the interface type that the receptacle will accept. The scoped name must denote a previously-defined non-component interface type.
- An <*identifier*> that names the receptacle in the scope of the component.

See the *CORBA Components* specification (Part 3), Component Model clause, “Receptacles” sub clause for further details.

5.17.4 Event Sources—publishers and emitters

An event source embodies the potential for the component to generate events of a specified type, and provides mechanisms for associating consumers with sources.

There are two categories of event sources, *publishers* and *emitters*. Both are implemented using event channels supplied by the container. An emitter can be connected to at most one consumer. A publisher can be connected through the channel to an arbitrary number of consumers, who are said to *subscribe* to the publisher event source. A component may exhibit zero or more emitters and publishers.

5.17.4.1 Publishers

Syntax

The syntax for an event publisher is as follows:

(124)<publishes_dcl> ::= “publishes” <scoped_name> <identifier>

A publisher declaration consists of the following elements:

- The keyword **publishes**.
- A <*scoped_name*> that denotes a previously-defined event type.
- An <*identifier*> that names the publisher event source in the scope of the component.

See the *CORBA Components* specification, Component Model clause, “Publisher” sub clause for further details.

5.17.4.2 Emitters

Syntax

The syntax for an emitter declaration is as follows:

(123)<emits_dcl> ::= “emits” <scoped_name> <identifier>

An emitter declaration consists of the following elements:

- The keyword **emits**.
- A <*scoped_name*> that denotes a previously-defined event type.
- An <*identifier*> that names the event source in the scope of the component.

See the *CORBA Components* specification, Component Model clause, “Emitters” sub clause for further details.

5.17.5 Event Sinks

An event sink embodies the potential for the component to receive events of a specified type. An event sink is, in essence, a special-purpose facet whose type is an event consumer. External entities, such as clients or configuration services, can obtain the reference for the consumer interface associated with the sink.

A component may exhibit zero or more consumers.

See the *CORBA Components* specification, Component Model clause, “Event Sinks” sub clause for further details.

Syntax

The syntax for an event sink declaration is as follows:

(125)<consumes_dcl> ::= “consumes” <scoped_name> <identifier>

An event sink declaration contains the following elements:

- The keyword **consumes**.
- A <*scoped_name*> that denotes a previously-defined event type.
- An <*identifier*> that names the event sink in the component’s scope.

See the *CORBA Components* specification, Component Model clause, “Event Sinks” sub clause for further details.

5.17.6 Basic and Extended Components

A component that satisfies the following properties is known as a *Basic Component*:

- It does not inherit from another component.
- Its declaration does not contain any provides statements.
- Its declaration does not contain any uses statements.
- Its declaration does not contain any publishes, emits, or consumes statements.

In effect a declaration of a *Basic Component* fits the pattern:

“component” <identifier> [<supported_interface_spec>]
“{“ {<attr_dcl> “;”}* “}”

A component that is not a *Basic Component* is referred to as an *Extended Component*.

5.18 Home Declaration

A home declaration describes an interface for managing instances of a specified component type.

5.18.1 Home

The salient characteristics of a home declaration are as follows:

- A home declaration must specify exactly one component type that it manages. Multiple homes may manage the same component type.
- A home declaration may specify a primary key type. Primary keys are values assigned by the application environment that uniquely identify component instances managed by a particular home. Primary key types must be value types

derived from **Components::PrimaryKeyBase**. There are more specific constraints placed on primary key types, which are specified in the *CORBA Components* specification, Component Model clause, “Primary key type constraints” sub clause.

- Home declarations may include any declarations that are legal in normal interface declarations.
- Home declarations support single inheritance from other home definitions, subject to a number of constraints that are described in the *CORBA Components* specification, Component Model clause, “Home inheritance” sub clause.
- Home declarations may specify a list of interfaces that the home supports.

Syntax

The syntax for a home definition is as follows:

(126) <home_dcl> ::= <home_header> <home_body>

<home_header> is described in “Home Header.”

<home_body> is described in “Home Body.”

5.18.2 Home Header

A <*home_header*> describes fundamental characteristics of a home interface.

Syntax

The syntax for a home header declaration is as follows:

(127) <home_header> ::= “home” <identifier>
[<home_inheritance_spec>]
[<supported_interface_spec>]
“manages” <scoped_name>
[<primary_key_spec>]

(128) <home_inheritance_spec> ::= “:” <scoped_name>

(129) <primary_key_spec> ::= “primarykey” <scoped_name>

A <*home_header*> consists of the following elements:

- The keyword **home**.
- An <*identifier*> that names the home in the enclosing name scope.
- An optional <*home_inheritance_spec*>, consisting of a colon “**:**” and a single <*scoped_name*> that denotes a previously defined home type.
- An optional <*supported_interface_spec*> that must denote one or more previously defined IDL interfaces.
- The keyword **manages**.
- A <*scoped_name*> that denotes a previously defined component type.
- An optional primary key definition, consisting of the keyword **primarykey** followed by a <*scoped_name*> that denotes a previously defined value type that is derived from the abstract value type **Components::PrimaryKeyBase.** Additional constraints on primary keys are described in the *CORBA Components* specification, Component Model clause, “Primary key type constraints” sub clause.

Details of semantics can be found in the *CORBA Components* specification, Component Model clause, “Homes” sub clause.

5.18.3 Home Body

(130) <home_body> ::= “{” <home_export>* “}”
(131) <home_export ::= <export>
| <factory_dcl> “;”
| <finder_dcl> “;”

5.18.3.1 Operation Declarations

A home body may include zero or more operation declarations, where the operation may be a *factory* operation, a *finder* operation, or a normal operation or attribute.

Factory operations

The syntax of a factory operation is as follows:

(132) <factory_dcl> ::= “factory” <identifier>
“(“ [<init_param_decls>] “)”
[<raises_expr>]

A factor operation declaration consists of the following elements:

- The keyword **factory**.
- An <identifier> that names the operation in the scope of the home declaration.
- An optional list of initialization parameters (<*init_param_decls*>) enclosed in parentheses.
- An optional <*raises_expr*> declaring exceptions that may be raised by the operation.

A factory declaration has an implicit return value of type reference to component.

See the *CORBA Components* specification, Component Model clause, “Factory operations” sub clause for further details.

Finder operations

The syntax of a finder operation is as follows:

(133) <finder_dcl> ::= “finder” <identifier>
“(“ [<init_param_decls>] “)”
[<raises_expr>]

A finder operation declaration consists of the following elements:

- The keyword **finder**.
- An identifier that names the operation in the scope of the storage home declaration.
- An optional list of initialization parameters (<*init_param_decls*>) enclosed in parentheses.
- An optional <*raises_expr*> declaring exceptions that may be raised by the operation.

A finder declaration has an implicit return value of type reference to component.

See the *CORBA Components* specification, Component Model clause, “Finder operations” sub clause for further details.

5.19 IDL3+ Grammar

The following description of IDL grammar extensions uses the same syntax motation that is used to describe OMG IDL in CORBA Core, IDL Syntax and Semantics clause. For reference, the following table lists the symbols used in this format and their meaning.

Table 5.12 - IDL EBNF Notation

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

5.19.1 Summary of IDL Grammar Extensions

The following table gathers all the new grammar rules supporting this specification. Those rules aim at completing the existing IDL grammar (“OMG IDL Syntax and Semantics” [IDL]).

The items that are in *italics-blue* are already described in the existing IDL grammar. When they appear here in the right part of a rule, they are considered as terminals. When they appear in the left part of a rule, they are extended by this specification.

IDL3+ Grammar Extensions

(134) **<definition>** **::=*<type_dcl>*** **“;”**
|*<const_dcl>* **“;”**
|*<except_dcl>* **“;”**
|*<interface>* **“;”**
|*<module>* **“;”**
|*<value>* **“;”**
|*<type_id_dcl>* **“;”**
|*<type_prefix_dcl>* **“;”**
|*<event>* **“;”**
|*<component>* **“;”**
|*<home_dcl>* **“;”**
|<porttype_dcl> “;”
|<connector> “;”
|<template_module> “;”

|<template_module_inst> “;”

(135) <porttype_dcl> ::=“porttype” <*identifier*> “{ ” <port_export>+ “}”

(136) <porttype_dcl> <port_export>::= <*provides_dcl*> “;”
|<*uses_dcl*> “;”
|<*attr_dcl*> “;”

(137) <port_dcl> ::={“port” | :mirrorport”} <*scoped_name*> <identifier>

(138) <*component_export*> ::=<*provides_dcl*> “;”
|<*uses_dcl*> “;”
|<*emits_dcl*> “;”
|<*publishes_dcl*> “;”
|<*consumes_dcl*> “;”
|<port_dcl> “;”
|<*attr_dcl*> “;”

(139) <connector> ::=<connector_header> “{” <connector_export>* “}”

(140) <connector_header> ::= “connector” <*identifier*>
[<connector_inherit_spec>]

(141) <connector_inherit_spec> ::= “:” <*scoped_name*>

(142) <connector_export> ::= <*provides_dcl*> “;”
|<*uses_dcl*> “;”
|<port_dcl> “;”
|<*attr_dcl*> “;”

(143) <template_module> ::= “module” <*identifier*> “<” <formal_parameters> “>” “{”
tpl_definition>” “}”

(144) <formal_parameters> ::= <formal_parameter> {“,” <formal_parameter>}*

(145) ::= <formal_parameter>::= <formal_parameter_type> <*identifier*>

(146) <formal_parameter_type>::= “typename”
|“interface” | “valuetype” | “eventtype”
|“struct” | “union” | “exception” | “enum” | “sequence”
|“const” <*const_type*>
|<*sequence_type*>

(147) <tpl_definition> ::= <*type_dcl*> “;”
|<*const_dcl*> “;”
|<*except_dcl*> “;”
|<*interface*> “;”
|<fixed_module> “;”
|<*value*> “;”
|<*type_id_dcl*> “;”
|<*type_prefix_dcl*> “;”
|<*event*> “;”
|<*component*> “;”
|<*home_dcl*> “;”
|<porttype_dcl> “;”
|<connector> “;”
|<template_module_ref> “;”

(148) <fixed_module> ::= “module” <*identifier*> “{” <fixed_definition>* “}”

(149) <fixed_definition> ::= <*type_dcl*> “;”
|<*const_dcl*> “;”
|<*except_dcl*> “;”

|<*interface*> “;”
|<fixed_module> “;”
|<*value*> “;”
|<*type_id_dcl*> “;”
|<*type_prefix_dcl*> “;”
|<*event*> “;”
|<*component*> “;”
|<*home_dcl*> “;”
|<porttype_dcl> “;”
|<connector> “;”

(150) <template_module_inst> ::= “module” <*scoped_name*> “<” <actual_parameters> “>” <*identifier*> “;”

(151) <actual_parameters> ::= <actual_parameter> “,” <actual_parameter>}*

(152) <actual_>parameter ::= <*type_spec*> “;”
|<*const_exp*> “;”

(153) <template_module_ref> ::= “alias” <*scoped_name*> “<” formal_parameter_names> “>” <*identifier*>

(154) <formal_parameter_names> ::= <*identifier*> { “,” <identifier>}*

Those rules are detailed in the following sub clauses.

5.19.2 New First-Level Constructs

The first rule extends the existing <**definition**> with the new first-level constructs that can be used natively or inside a module, namely:

- port type declarations,
- connector declarations,
- template module declarations,
- template module instnatiations.

Those new constructs are detailed in the following sub clauses.

(155) <*definition*> ::= <*type_dcl*> “;”
|<*const_dcl*> “;”
|<*except_dcl*> “;”
|<*interface*> “;”
|<*module*> “;”
|<*value*> “;”
|<*type_id_dcl*> “;”
|<*type_prefix_dcl*> “;”
|<*event*> “;”
|<*component*> “;”
|<*home_dcl*> “;”
|<porttype_dcl> “;”
|<connector> “;”
|<template_module> “;”
|<template_module_inst> “;”

5.19.3 IDL Extensions for Extended Ports

5.19.3.1 Port Type Declarations

The following rules allow port type declarations:

(156) **<porttype_dcl> ::= “porttype” <*identifier*> “{” <port_export>+ “}”**

(157) **<port_export> ::= <*provides_dcl*> “;”**
|<*uses_dcl*> “;”
|<*attr_dcl*> “;”

A port type declaration is made of:

- the porttype keyword,
- an identifier for the port type name,
- the list, of provided and/or used basic ports and attributes, that constitutes the extended port.

5.19.3.2 Extended Port Declarations

The following rules allow port declarations:

(158) **<port_dcl> ::= {“port” | “mirrorport” } <*scoped_name*> <identifier>**

(159) **<component_export> ::= <*provides_dcl*> “;”**
|<*uses_dcl*> “;”
|<*emits_dcl*> “;”
|<*publishes_dcl*> “;”
|<*consumes_dcl*> “;”
|<port_dcl> “;”
|<*attr_dcl*> “;”

An extended port declaration comprises:

- the **port** or **mirrorport** keyword,
- the name of a previously defined port type,
- the identifier for the port.

The existing <**component_export**> is modified so that such a port declaration can be used to add an extended port to a component.

5.19.4 IDL Extensions for Connectors

The following rules allow connector declarations:

(160) **<connector> ::= <connector_header> “{” <connector_export>* “}”**

(161) **<connector_header> ::= “connector” <*identifier*> [<connector_inherit_spec>]**

(162) **<connector_inherit_spec>::= “:” <*scoped_name*>**

(163) **<connector_export>::= <*provides_dcl*> “;”**
|<*uses_dcl*> “;”
|<port_dcl> “;”

|<*attr_dcl*> “;”

A connector is defined by its header and its body.

A connector header comprises:

- the keyword **connector**,
- an identifier for the connector,
- an optional inheritance specification, consisting of a colon and a single scoped name that must denote a previously-defined connector.

A connector body may comprise:

- facet declarations,
- receptable declarations,
- extended port declarations,
- attribute declarations.

5.19.5 IDL Extensions for Template Modules

5.19.5.1 Template Module Declarations

The following rules allow template module declarations.

(164) **<template_module> ::= “module” <*identifier*> “<” <formal_parameters> “>” “{“ <tpl_definition>+ “}“**

(165) **<formal_parameters> ::= <formal_parameter> {“,” <formal_parameter>}***

(166) **<formal_parameter> ::= <formal_parameter_type> <*identifier*>**

(167) **<formal_parameter_type> ::= “typename”**
| “*interface*”| “*valuetype*”| “*eventtype*”
| “*struct*”| “*union*”| “*exception*”| “*enum*”| “*sequence*”
| “*const*” <*const_type*>
| <*sequence_type*> “;”

(168) **<tpl_definition> ::= <*type_dcl*>**
|<*const_dcl*> “;”
|<*except_dcl*> “;”
|<*interface*> “;”
|<fixed_module> “;”
|<*value*> “;”
|<*type_id_dcl*> “;”
|<*type_prefix_dcl*> “;”
|<*event*> “;”
|<*component*> “;”
|<*home_dcl*> “;”
|<porttype_dcl> “;”
|<connector> “;”
|<template_module_ref> “;”

(169) **<fixed_module> ::= “module” <*identifier*> “{” <fixed_definition>+“}”**

(170) **<fixed_definition> ::= <*type_dcl*> “;”**
|<*const_dcl*> “;”
|<*except_dcl*> “;”
|<*interface*> “;”
|<*fixed_module*> “;”
|<*value*> “;”
|<*type_id_dcl*> “;”
|<*type_prefix_dcl*> “;”
|<*event*> “;”
|<*component*> “;”
|<*home_dcl*> “;”
|<porttype_dcl> “;”
|<connector> “;”

A template module specification comprises:

- the **module** keyword
- an identifier for the module name
- the specification of the template parameters between angular brackets, each of those template parameters consisting of:
  - a type classifier, which can be:
    - **typename**, to indicate that any valid type can be passed as parameter
    - **interface**, **valuetype**, **eventtype**, **struct**, **union**, **exception**, **enum**, **sequence** to indicate that a more restricted type must be passed as parameter
    - a constant type, to indicate that a constant of that type must be passed as parameter
    - a sequence type declaration, to indicate that a compliant sequence type must be passed as parameter (the formal parameters of that sequence must appear previously in the module list of formal parameters)
  - an identifier for the formal parameter
- the module body, which may contain declarations for port types and/or connectors, other template module references, as well as all that previously made a classical module body (that last part is named <**fixed_module**> in the grammar).[2]

A template module cannot be re-opened (as opposed to a classical one).

5.19.5.2 Template Module Instantiations

The following rules allow template module instantiations:

(171) **<template_module_inst> ::= “module” <*scoped_name*> “<” <actual_parameters> “>” <*identifier*> “;”**

(172) **<actual_parameters> ::= <actual_parameter>{“,” <actual_parameter>}***

(173) **<actual_parameter> ::= <type_spec>**
|<*const_exp*>

A module template instantiation consists in providing values to the template parameters and a name to the resulting module. Once instantiated, the module is exactly as a classical module.

---

2. Note that this imples that a template module cannot contain another template module.

The provided values must fit with the parameter specification as described in the previous sub clause. In particular, if the template parameter is of type “sequence type declaration,” then an instantiated compliant sequence must be passed.

5.19.5.3 References to a Template Module

The following rules allow referencing template modules:

(174) <template_module_ref > ::= “alias” <*scoped_name*> “<” <formal_parameter_names> “>” <*identifier*>

(3) <formal_parameter_names>::= <*identifier*> {“,” <*identifier*>}*

An alias directive allows to reference an existing template module inside a template module definition.

This directive allows to provide an alias name (which can be identical to the template module name) and the list of formal parameters to be used for the referenced module instantiation. Note that that list must be a subset of the formal parameters of the embedding module.

When the embedding module will be instantiated, then the referenced module will be instantiated in the scope of the embedding one (i.e., as a submodule).

5.19.6 Summary of New IDL Keywords

The following table gathers all new keywords introduced by this specification.

Table 5.13 - New IDL Keywords

alias	connector	mirrorport	port	porttype	typename

As all IDL keywords, they are now reserved and thus may not be used otherwise, unless escaped with a leading underscore.

5.20 CORBA Module

Names defined by the CORBA specification are in a module named CORBA. In an IDL specification, however, IDL keywords such as **Object** must not be preceded by a “**CORBA::**” prefix. Other interface names such as **TypeCode** are not IDL keywords, so they must be referred to by their fully scoped names (e.g., **CORBA::TypeCode**) within an 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. Except for **CORBA::TypeCode**, the file **orb.idl** must be included in IDL files that use names defined in the **CORBA** module. IDL files that use **CORBA::TypeCode** may obtain its definition by including either the file **orb.idl** or the file **TypeCode.idl**.

The exact contents of **TypeCode.idl** are implementation dependent. One possible implementation of **TypeCode.idl** may be:

```
// PIDL
#ifndef _TYPECODE_IDL_
#define _TYPECODE_IDL_
#pragma prefix "omg.org"
module CORBA {
    interface TypeCode;
};
#endif // _TYPECODE_IDL_
```

For IDL compilers that implicitly define **CORBA::TypeCode**, **TypeCode.idl** could consist entirely of a comment as shown below:

```
// PIDL
// CORBA::TypeCode implicitly built into the IDL compiler
// Hence there are no declarations in this file
```

Because the compiler implicitly contains the required declaration, this file meets the requirement for compliance.

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 the CORBA 3.2 specification, Sub clause 14.7.5.3 “The Version Pragma”):

```
#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.

5.21 Names and Scoping

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. 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
        );
    };
};
```

5.21.1 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 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 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 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 IDL files correspond to an absolute **ScopedName** in the Interface Repository. (See the CORBA 3.2 specification, Sub clause 14.5.1 "Supporting Type Definitions.")

Inheritance causes all identifiers defined in base interfaces, both direct and indirect, to be visible in derived interfaces. Such identifiers are considered to be semantically the same as the original definition. Multiple paths to the same original identifier (as results from the diamond shape in Figure 7.1 on page 23) do not conflict with each other.

Inheritance introduces multiple global 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.

5.21.2 Scoping Rules and Name Resolution

Contents of an entire 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
- eventtype
- component
- home

The scope for module, interface, valuetype, struct, exception, eventtype, component, and home 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 a 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.

Only the first identifier in a qualified name is introduced into the current scope. This is illustrated by **Inner1::S1** in the example above, which introduces "**Inner1**" into the scope of "**Inner2**" but does not introduce "**S1**." A qualified name of the form "**::X::Y::Z**" does not cause "**X**" to be introduced, but a qualified name of the form "**X::Y::Z**" does.

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 5.11.2, Constructed Types 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.**

5.21.3 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, interface or valuetype, or within the scope of a derived interface or valuetype. For example:

```
typedef short TempType;          // Scope of TempType begins here

module M {
    typedef string ArgType;     // Scope of ArgType begins here
    struct S {
        ::M::ArgType a1;         // Nothing introduced here
        M::ArgType a2;           // M introduced here
        ::TempType temp;        // Nothing introduced here
    };                  // Scope of (introduced) M ends here
    // ...
};                       // Scope of ArgType ends here

// Scope of global TempType ends here (at end of file)
```

The scope of an introduced type name is from the point of introduction to the end of its enclosing scope.

However, if a *type* name is *introduced* into a scope that is nested in a non-module scope definition, its *potential* scope extends over all its enclosing scopes out to the enclosing non-module scope. (For types that are defined outside an inon-module scope, the scope and the potential scope are identical.) For example:

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

    interface A {
        struct S {
            struct T {
                ArgType x[I];   // ArgType and I introduced
                long y;         // a new y is defined, the existing Y
                                // is not used
            } m;
        };
        typedef string ArgType;     // Error: ArgType redefined
        enum I { I1, I2 };          // Error: I redefined
        typedef short Y;            // OK
    };   // Potential scope of  ArgType and I ends here

    interface B : A {
        typedef long ArgType  // OK, redefined in derived interface
        struct S {            // OK, redefined in derived interface
            ArgType x;        // x is a long
            A::ArgType y;     // y is a string
        };
    };
};
```

A type may not be redefined within its scope or potential scope, as shown in the preceding example. This rule prevents type names from changing their meaning throughout a non-module scope definition, and ensures that reordering of definitions in the presence of introduced types does not affect the semantics of a specification.

Note that, in the following, the definition of **M::A::U::I** is legal because it is outside the potential scope of the I introduced in the definition of **M::A::S::T::ArgType**. However, the definition of **M::A::I** is still illegal because it is within the potential scope of the I introduced in the definition of **M::A::S::T::ArgType**.

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

    interface A {
        struct S {
            struct T {
                ArgType x[I];    // ArgType and I introduced
            } m;
        };
        struct U {
            long I;              // OK, I is not a type name
        };
        enum I { I1, I2 };      // Error: I redefined
    }; // Potential scope of  ArgType and I ends here
};
```

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
    };
}
```

6 Value Type Semantics

6.1 Overview

Objects, more specifically, interface types that objects support, are defined by an IDL interface, allowing arbitrary implementations. There is great value, which is described in great detail elsewhere, in having a distributed object system that places almost no constraints on implementations.

However there are many occasions in which it is desirable to be able to pass an object by value, rather than by reference. This may be particularly useful when an object's primary "purpose" is to encapsulate data, or an application explicitly wishes to make a "copy" of an object.

The semantics of passing an object by value are similar to that of standard programming languages. The receiving side of a parameter passed by value receives a description of the "state" of the object. It then instantiates a new instance with that state but having a separate identity from that of the sending side. Once the parameter passing operation is complete, no relationship is assumed to exist between the two instances.

Because it is necessary for the receiving side to instantiate an instance, it must necessarily know something about the object's state and implementation.

Value types provide semantics that bridge between CORBA structs and CORBA interfaces:

- They support description of complex state (i.e., arbitrary graphs, with recursion and cycles).
- Their instances are always local to the context in which they are used (because they are always copied when passed as a parameter to a remote call).
- They support both public and private (to the implementation) data members.
- They can be used to specify the state of an object implementation (i.e., they can support an interface).
- They support single inheritance (of **valuetype**) and can support an **interface**.
- They may be also be **abstract**.

6.2 Architecture

The basic notion is relatively simple. A **value type** is, in some sense, half way between a "regular" IDL interface type and a struct. The use of a value type is a signal from the designer that some additional properties (state) and implementation details be specified beyond that of an interface type. Specification of this information puts some additional constraints on the implementation choices beyond that of interface types. This is reflected in both the semantics specified herein, and in the language mappings.

An essential property of value types is that their implementations are always local. That is, the explicit use of value type in a concrete programming language is always guaranteed to use a local implementation, and will not require a remote call. They have no identity (their value is their identity) and they are not "registered" with the ORB.

There are two kinds of value types, concrete (or stateful) value types, and abstract (stateless) ones. As explained below the essential characteristics of both are the same. The differences between them result from the differences in the way they are mapped in the language mappings. In this specificaiton the semantics of value types apply to both kinds, unless specifically stated otherwise.

Concrete (stateful) values add to the expressive power of (IDL) structs by supporting:

- Single derivation (from other value types).

- Supports a single non-abstract interface.
- Arbitrary recursive value type definitions, with sharing semantics providing the ability to define lists, trees, lattices, and more generally arbitrary graphs using value types.
- Null value semantics.

When an instance of such a type is passed as a parameter, the sending context marshals the state (data) and passes it to the receiving context. The receiving context instantiates a new instance using the information in the GIOP request and unmarshals the state. It is assumed that the receiving context has available to it an implementation that is consistent with the sender's (i.e., only needs the state information), or that it can somehow download a usable implementation. Provision is made in the on-the-wire format to support the carrying of an optional call back object (**CodeBase**) to the sending context, which enables such downloading when it is appropriate.

It should be noted that it is possible to define a concrete value type with an empty state as a degenerate case.

6.2.1 Abstract Values

Value types may also be abstract. They are called abstract because an abstract value type may not be instantiated. Only concrete types derived from them may be actually instantiated and implemented. Their implementation, of course, is still local. However, because no state information may be specified (only local operations are allowed), abstract value types are not subject to the single inheritance restrictions placed upon concrete value types. Essentially they are a bundle of operation signatures with a purely local implementation. This distinction is made clear in the language mappings for abstract values.

Note that a concrete value type with an empty state is not an abstract value type. They are considered to be stateful, may be instantiated, marshaled, and passed as actual parameters. Consider them to be a degenerate case of stateful values.

6.2.2 Operations

Operations defined on a value type specify signatures whose implementation can only be local. Because these operations are local, they must be directly implemented by a body of code in the language mapping (no proxy or indirection is involved).

The language mappings of such operations require that instances of value types passed into and returned by such local methods are passed by reference (programming language reference semantics, not CORBA object reference semantics) and that a copy is not made. Note, such a (local) invocation is not a CORBA invocation. Hence it is not mediated by the ORB, although the API to be used is specified in the language mapping.

The (copy) semantics for instances of value type are only guaranteed when instances of these value types are passed as a parameter to an operation defined on a CORBA interface, and hence mediated by the ORB. If an instance of a value type is passed as a parameter to a method of another value type in an invocation, then this call is a "normal" programming language call. In this case both of the instances are local programming language constructs. No CORBA style copy semantics are used and programming language reference semantics apply.

Operations on the value type are supported in order to guarantee the portability of the client code for these value types. They have no representation on the wire and hence no impact on interoperability.

6.2.3 Value Type vs. Interfaces

By default value types are not CORBA Objects. In particular, instances of value types do not inherit from **CORBA::Object** and do not support normal object reference semantics. However it is always possible to explicitly declare that a given value type supports an interface type. In this case instances of the type may support CORBA object reference semantics (if they are registered with the ORB using an object adapter).

6.2.4 Parameter Passing

This sub clause describes semantics when a value instance is passed as parameter in a CORBA invocation. It does not deal with the case of calling another non-CORBA (i.e., local) programming method, which happens to have a parameter of the same type.

6.2.4.1 Value vs. Reference Semantics

Determination of whether a parameter is to be passed by value or reference is made by examining the parameter's formal type (i.e., the signature of the operation it is being passed to). If it is a value type, then it is passed by value. If it is an ordinary interface, then it is passed by reference (the case today for all CORBA objects). This rule is simple and consistent with the handling of the same situation in recursive state definitions or in structs.

In the case of abstract interfaces, the determination is made at runtime. See Semantics of Abstract Interfaces on page 93 for a description of the rules.

6.2.4.2 Sharing Semantics

In order to be expressive enough to describe arbitrary graphs, lattice, trees, etc., value types support sharing and null semantics. Instances of a value type can be shared by others across or within other instances. They can also be null. This is unlike other IDL data types such as structs, unions, and sequences that can never be shared. The sharing of values within and between the parameters to an operation is preserved across an invocation; that is, the graph that is reconstructed in the receiving context is structurally isomorphic to the sending context's.

6.2.4.3 Identity Semantics

When an instance of the value type is passed as a parameter to an operation of a non-local interface, the effect in all cases shall be as if an independent copy of the instance is instantiated in the receiving context. While certain implementation optimizations are possible the net effect shall be as if the copy is a separate independent entity and there is no explicit or implicit sharing of state. This applies to all valuetypes involved in the invocation, including those embedded in other IDL datatypes or in an any. This notional copying occurs twice, once for in and inout parameters when the invocation is initiated, and once again for inout, out, and return parameters when the invocation completes. Optimization techniques such as copy on write, etc. must make sure that the semantics of copying as described above is preserved.

6.2.4.4 Any parameter type

When an instance of a value type is passed to an **any**, as with all cases of passing instances to an **any**, it is the responsibility of the implementor to insert and extract the value according to the language mapping specification.

6.2.5 Substitutability Issues

The substitutability requirements for CORBA require the definition of what happens when an instance of a derived value type is passed as a parameter that is declared to be a base value type or an instance of a value type that supports an interface is passed as a parameter that is declared as the interface type.

There are three cases to consider: the parameter type is a regular interface, the parameter type is an abstract interface, and the parameter type is a value type.

6.2.5.1 Value instance -> Interface type

A value type that supports a regular interface is not a subtype of that interface, and hence cannot be substituted for that interface in an invocation parameter. In this case an object reference corresponding to the value type instance that has been registered with the ORB must be obtained and this object reference must be used as the actual parameter. Different language mappings provide different facilities to aid in such parameter passing.

6.2.5.2 Value Instance -> Abstract interface type

A value type that supports an abstract interface is a subtype of that interface, and can be substituted for that interface in an invocation parameter.

6.2.5.3 Value instance -> Value type

In this case the receiving context is expecting to receive a value type. If the receiving context currently has the appropriate implementation class, then there is no problem.

If the receiving context does not currently hold an implementation with which to reconstruct the original type, then the following algorithm is used to find such an implementation:

1. ***Load*** - Attempt to load (locally in C/C++, possibly remotely in Java and other “portable” languages) the real type of the object (with its methods). If this succeeds, OK.

2. ***Truncate*** - Truncate the type of the object to the base type (if specified as **truncatable** in the IDL). Truncation can never lead to faulty programs because, from a structural point view base types structurally subsume a derived type and an object created in the receiving context bears no relationship with the original one. However, it might be semantically puzzling, as the derived type may completely re-interpret the meaning of the state of the base. For that reason a derived value needs to indicate if it is safe to truncate to its immediate non-abstract parent.

3. ***Raise Exception*** - If none of these work or are possible, then raise the NO_IMPLEMENT exception with standard minor code 1.

Truncatability is a transitive property.

Example

```
valuetype EmployeeRecord {  // note this is not a CORBA::Object
    // state definition
    private string name;
    private string email;
    private string SSN;
    // initializer
    factory init(in string name, in string SSN);
};
```

```
valuetype ManagerRecord: truncatable EmployeeRecord {
    // state definition
    private sequence<EmployeeRecord> direct_reports;
};
```

6.2.6 Widening/Narrowing

As has been described above, value type instances may be widened/narrowed to other value types. Each language mapping is responsible for specifying how these operations are made available to the programmer.

Narrowing from an interface type instance to a value type instance is not allowed. If the interface designer wants to allow the receiving context to create a local implementation of the value type (i.e., a value representing the interface), an operation that returns the appropriate value type may be defined.

6.2.7 Value Base Type

All value types have a conventional base type called **ValueBase**. This is a type, which fulfills a role that is similar to that played by **Object**. Conceptually it supports the common operations available on all value types. See Value Base Type Operation for a description of those operations. In each language mapping **ValueBase** will be mapped to an appropriate base type that supports the marshaling/unmarshaling protocol as well as the model for custom marshaling.

The mapping for other operations, which all value types must support, such as getting meta information about the type, may be found in the specifics for each language mapping.

6.2.8 Life Cycle issues

Value type instances are always local to their creating context. For example, in a given language mapping an instance of a value type is always created as a local “language” object with no POA semantics attached to it initially.

When passed using a CORBA invocation, a copy of the value is made in the receiving context and that copy starts its life as a local programming language entity with no POA semantics attached to it.

If a value type supports an ordinary interface type, its instances may also be passed by reference when the formal parameter type is an interface type (see Parameter Passing on page 79). In this case they behave like ordinary object implementations and must be associated with a POA policy and also be registered with the ORB (e.g., **POA::activate_object()** before they can be passed by reference. Not registering the value as a CORBA object and/or not associating an appropriate policy with it results in an exception when trying to use it as a remote object, the “normal” behavior. The exception raised shall be OBJECT_NOT_EXIST with standard minor code 1.

6.2.8.1 Creation and Factories

When an instance of a value type is received by the ORB, it must be unmarshaled and an appropriate factory for its actual type found in order for the new instance to be created. The type is encoded by the RepositoryID, which is passed over the wire as part of an invocation. The mapping between the type (as specified by the RepositoryID) and the factory is language specific. In certain languages it may be possible to specify default policies that are used to find the factory, without requiring that specific routines be called. In others the runtime and/or generated code may have to explicitly specify the mapping on a per type basis. In others a combination may be used. In any event the ORB implementation is responsible for maintaining this mapping See Language Specific Value Factory Requirements on page 83 for more details on the requirements for each language mapping. Value box types do not need or use factories.

6.2.9 Security Considerations

The addition of value types has few impacts on the CORBA security model. In essence, the security implications in defining and using value types are similar to those involved with the use of IDL structs. Instances of value types are mapped to local, concrete programming language constructs. Except for providing the marshaling mechanisms, the ORB is not directly involved with accessing value type implementations. This specification is mostly about two things: how value types manifest themselves as concrete programming language constructs and how they are transmitted.

To see this consider how value types are actually used. The IDL definition of a value type in conjunction with a programming language mapping is used to generate the concrete programming language definitions for that type.

Let us consider its life cycle. In order to use it, the programmer uses the mechanisms in the programming language to instantiate an instance. This is instance is a local programming language construct. It is not “registered” with the ORB, object adapter, etc. The programmer may manipulate this programming construct just like any other programming language construct. So far there are no security implications. As long as no ORB-mediated invocations are made, the programmer may manipulate the construct. Note, this includes making “local,” non ORB-mediated calls to any locally implemented operations. Any assignments to the construct are the responsibility of the programmer and have no special security implications.

Things get interesting when the program attempts to pass one of these constructs through an orb-mediated invocation (i.e., calls a stub that uses it as a parameter type, or uses the DII). There are two cases to consider: 1) Value as Value and 2) Value as Object Reference.

6.2.9.1 Value as Value

The formal type of the parameter is a value. This case is no different from using any other kind of a value (long, string, struct) in a CORBA invocation, with respect to security. The value (data) is marshaled and delivered to the receiving context. On the receiving context, the knowledge of the type is used (at least implicitly) to find the factory to create the correct local programming language construct. The data is then unmarshaled to fill in the newly created construct. This is similar to using other values (longs, strings, structs) except that the knowledge of the factory is not “built-in” to the ORB’s skeleton/DSI engine.

6.2.9.2 Value as Object Reference

The formal type of the parameter is an interface type that is supported by a value. The program must have “registered” the value with an object adapter and is really using the returned object reference (see for the specific rules.) Thus this case “reduces” to a regular CORBA invocation, using a regular object reference. An IOR is passed to the receiving context. All the “normal” security considerations apply. From the point of view of the receiving context, the IOR is a “normal” object reference. No “special” rules, with respect to security or otherwise, apply to it. The fact that it is ultimately a reference to an implementation that was created from instantiating and registering a value type implementation is not relevant.

In both of these cases, security considerations are involved with the decision to allow the ORB-mediated invocation to proceed. The fact that a value type is involved is not material.

6.3 Standard Value Box Definitions

For some CORBA-defined types for which preservation of sharing and transmission of nulls are likely to be important, the following value box type definitions are added to the CORBA module.

```
module CORBA {
    valuetype StringValue string;
    valuetype WStringValue wstring;
};
```

6.4 Language Mappings

6.4.1 General Requirements

A concrete value is mapped to a concrete usable “class” construct in each programming language, plus possibly some helper classes where appropriate. In Java, C++, and Smalltalk this is a real concrete class. In C it is a struct.

An abstract value is mapped to some sort of an abstract construct--an interface in Java, and an abstract class with pure virtual function members in C++.

Tools that implement the language mapping are free to “extend” the implementation classes with “extra” data members and methods. When an instance of such a class is used as a parameter, only the portions that correspond directly to the IDL declaration, are marshaled and delivered to the receiving context. This allows freedom of implementations while preserving the notion of contract and type safety in IDL.

6.4.2 Language Specific Marshaling

Each language mapping defines an appropriate marshaling/unmarshaling API and the entry point for custom marshaling/unmarshaling.

6.4.3 Language Specific Value Factory Requirements

Each language mapping specifies the algorithm and means by which RepositoryIDs are used to find the appropriate factory for an instance of a value type so that it may be created as it is unmarshaled “off the wire.”

It is desirable, where it makes sense, to specify a “default” policy for automatically using RepositoryIDs that are in common formats to find the appropriate factory. Such a policy can be thought of as an implicit registration.

Each language mapping specifies how and when the registration occurs, both explicit and implicit. The registration must occur before an attempt is made to unmarshal an instance of a value type. If the ORB is unable to locate and use the appropriate factory, then a MARSHAL exception with standard minor code 1 is raised.

Because the type of the factory is programming language specific and each programming language platform has different policies, the factory type is specified as **native**. It is the responsibility of each language mapping to specify the actual programming language type of the factory.

```
module CORBA {

    // IDL
    native ValueFactory;
};
```

6.4.4 Value Method Implementation

The mapped class must support method bodies (i.e., code) that implement the required IDL operations. The means by which this association is accomplished is a language mapping “detail” in much the same way that an IDL compiler is.

6.5 Custom Marshaling

Value types can override the default marshaling/unmarshaling model and provide their own way to encode/decode their state. Custom marshaling is intended to be used to facilitate integration of existing “class libraries” and other legacy systems. It is explicitly not intended to be a standard practice, nor used in other OMG specifications to avoid “standard ORB” marshaling.

The fact that a value type has some custom marshaling code is declared explicitly in the IDL. This explicit declaration has two goals:

- ***Type safety*** - stubs and skeleton can know statically that a given type is custom marshaled and can then do a sanity check on what is coming over the wire.
- ***efficiency*** - for value types that are not custom marshaled no run time test is necessary in the marshaling code.

If a custom marshaled value type has a state definition, the state definition is treated the same as that of a non custom value type for mapping purposes (i.e., the fields show up in the same fashion in the concrete programming language). It is provided to help with application portability.

A custom marshaled value type is always a stateful value type.

```
// Example IDL

custom valuetype T {
    // optional state definition
        ...
};
```

Custom value types can never be safely truncated to base (i.e., they always require an exact match for their RepositoryId in the receiving context).

Once a value type has been marked as custom, it needs to provide an implementation that marshals and unmarshals the valuetype. The marshaling code encapsulates the application code that can marshal and unmarshal instances of the value type over a stream using the CDR encoding. It is the responsibility of the implementation to marshal the state of all of its base types.

The following sub clauses define the operations and streams that are used for custom marshaling.

6.5.1 Implementation of Custom Marshaling

Once a value type has been marked as custom, an implementation of the custom marshaling code must be provided. This is specified by providing a concrete implementation of an abstract value type, **CustomMarshal**, as part of the implementation of the value type. **CustomMarshal** encapsulates the application code that can marshal and unmarshal instances of the value type over a stream using the CDR encoding.

The following IDL defines the interfaces that are used to support the definition and use of custom marshaling.

```
module CORBA {
    abstract valuetype CustomMarshal {
        void marshal (in DataOutputStream os);
        void unmarshal (in DataInputStream is);
    };
};
```

CustomMarshal is an abstract value type that is meant to be used by the ORB, not the user. Semantically it is treated as a custom valuetype's implicit base class, although the custom valuetype does not actually inherit it in IDL. The implementor of a custom value type provides an implementation of the **CustomMarshal** operations. The manner in which this is done is specified for each language mapping. Each custom marshaled value type has its own implementation. The interface is exposed in the CORBA module so that the implementor can use the skeletons generated by the IDL compiler as the basis for the implementation. Hence there is no need for the application to acquire a reference to a Stream.

Note that while nothing prevents a user from writing IDL that inherits from **CustomMarshal**, doing so will not make the type custom, nor will it cause the ORB to treat it as custom.

The implementation requirements of the streaming mechanism require that the implementations must be local since local memory addresses (i.e., the marshal buffers) have to be manipulated.

6.5.2 Marshaling Streams

The streams used for marshaling are defined below. They are responsible for marshaling and demarshaling the data that makes up a custom value in CDR format.

```
module CORBA {

    typedef sequence<any> AnySeq;
    typedef sequence<boolean> BooleanSeq;
    typedef sequence<char> CharSeq;
    typedef sequence<wchar> WCharSeq;
    typedef sequence<octet> OctetSeq;
    typedef sequence<short> ShortSeq;
    typedef sequence<unsigned short> UShortSeq;
    typedef sequence<long> LongSeq;
    typedef sequence<unsigned long> ULongSeq;
    typedef sequence<long long> LongLongSeq;
    typedef sequence<unsigned long long> ULongLongSeq;
    typedef sequence<float> FloatSeq;
    typedef sequence<double> DoubleSeq;
    typedef sequence<long double> LongDoubleSeq;
    typedef sequence<string> StringSeq;
    typedef sequence<wstring> WStringSeq;

    exception BadFixedValue {
        unsigned long offset;
    };

    abstract valuetype DataOutputStream {
        void write_any(in any value);
```

```
void write_boolean(in boolean value);
void write_char(in char value);
void write_wchar(in wchar value);
void write_octet(in octet value);
void write_short(in short value);
void write_ushort(in unsigned short value);
void write_long(in long value);
void write_ulong(in unsigned long value);
void write_longlong(in long long value);
void write_ulonglong(in unsigned long long value);
void write_float(in float value);
void write_double(in double value);
void write_longdouble(in long double value);
void write_string(in string value);
void write_wstring(in wstring value);
void write_Object(in Object value);
void write_Abstract(in AbstractBase value);
void write_Value(in ValueBase value);
void write_TypeCode(in TypeCode value);

void write_any_array(
    in AnySeq seq,
    in unsigned long offset,
    in unsigned long length
);
void write_boolean_array(
    in BooleanSeq seq,
    in unsigned long offset,
    in unsigned long length
);
void write_char_array(
    in CharSeq seq,
    in unsigned long offset,
    in unsigned long length
);
void write_wchar_array(
    in WCharSeq seq,
    in unsigned long offset,
    in unsigned long length
);
void write_octet_array(
    in OctetSeq seq,
    in unsigned long offset,
    in unsigned long length
);
void write_short_array(
    in ShortSeq seq,
    in unsigned long offset,
    in unsigned long length
);
void write_ushort_array(
```

```
            in UShortSeq seq,
            in unsigned long offset,
            in unsigned long length
        );
        void write_long_array(
            in LongSeq seq,
            in unsigned long offset,
            in unsigned long length
        );
        void write_ulong_array(
            in ULongSeq seq,
            in unsigned long offset,
            in unsigned long length
        );
        void write_ulonglong_array(
            in ULongLongSeq seq,
            in unsigned long offset,
            in unsigned long length
        );
        void write_longlong_array(
            in LongLongSeq seq,
            in unsigned long offset,
            in unsigned long length
        );
        void write_float_array(
            in FloatSeq seq,
            in unsigned long offset,
            in unsigned long length
        );
        void write_double_array(
            in DoubleSeq seq,
            in unsigned long offset,
            in unsigned long length
        );

        void write_long_double_array(
            in LongDoubleSeq seq,
            in unsigned long offset,
            in unsigned long length
        );

        void write_fixed(
            in any fixed_value
        ) raises (BadFixedValue);
        void write_fixed_array(
            in AnySeq seq,
            in unsigned long offset,
            in unsigned long length
        ) raises (BadFixedValue);
    };
```

```
abstract valuetype DataInputStream {
    any read_any();
    boolean read_boolean();
    char read_char();
    wchar read_wchar();
    octet read_octet();
    short read_short();
    unsigned short read_ushort();
    long read_long();
    unsigned long read_ulong();
    long long read_longlong();
    unsigned long long read_ulonglong();
    float read_float();
    double read_double();
    long double read_longdouble();
    string read_string();
    wstring read_wstring();
    Object read_Object();
    AbstractBase read_Abstract();
    ValueBase read_Value();
    TypeCode read_TypeCode();

    void read_any_array(
        inout AnySeq seq,
        in unsigned long offset,
        in unsigned long length
    );
    void read_boolean_array(
        inout BooleanSeq seq,
        in unsigned long offset,
        in unsigned long length
    );
    void read_char_array(
        inout CharSeq seq,
        in unsigned long offset,
        in unsigned long length
    );
    void read_wchar_array(
        inout WCharSeq seq,
        in unsigned long offset,
        in unsigned long length
    );
    void read_octet_array(
        inout OctetSeq seq,
        in unsigned long offset,
        in unsigned long length
    );
    void read_short_array(
        inout ShortSeq seq,
        in unsigned long offset,
        in unsigned long length
```

```
);
void read_ushort_array(
      inout UShortSeq seq,
      in unsigned long offset,
      in unsigned long length
);
void read_long_array(
      inout LongSeq seq,
      in unsigned long offset,
      in unsigned long length
);
void read_ulong_array(
      inout ULongSeq seq,
      in unsigned long offset,
      in unsigned long length
);
void read_ulonglong_array(
      inout ULongLongSeq seq,
      in unsigned long offset,
      in unsigned long length
);
void read_longlong_array(
      inout LongLongSeq seq,
      in unsigned long offset,
      in unsigned long length
);
void read_float_array(
      inout FloatSeq seq,
      in unsigned long offset,
      in unsigned long length
);
void read_double_array(
      inout DoubleSeq seq,
      in unsigned long offset,
      in unsigned long length
);

void read_long_double_array(
      inout DoubleSeq seq,
      in unsigned long offset,
      in unsigned long length
);
any read_fixed(
      in unsigned short digits,
      in short scale
) raises (BadFixedValue);
void read_fixed_array(
      inout AnySeq seq,
      in unsigned long offset,
      in unsigned long length,
      in unsigned short digits,
```

```
            in short scale
        ) raises (BadFixedValue);
    };
};
```

Note that the Data streams are abstract value types. This ensures that their implementation will be local, which is required in order for them to properly flatten and encode nested value types.

The **read**_ operations that have an inout parameter named seq are expected to extend the sequence to fit the read value.

The ORB (i.e., the CDR encoding engine) is responsible for actually constructing the value's encoding. The application marshaling code merely calls the above operations. The details of writing the value tag, header information, end tag(s) are specifically not exposed to the application code. In particular the size of the custom data is not written by the application. This guarantees that the custom marshaling (and unmarshaling code) cannot corrupt the other parameters of the call.

If an inconsistency is detected, then the standard system exception MARSHAL is raised.

A possible implementation might have the engine determine that a custom marshal parameter is "next." It would then write the value tag and other header information and then return control back to the application defined marshaling policy, which would do the marshaling by calling the **DataOutputStream** operations to write the data as appropriate. (Note the stream takes care of breaking the data into chunks, if necessary.) When control was returned back to the engine, it performs any other cleanup activities to complete the value type, and then proceeds onto the next parameter. How this is actually accomplished is an implementation detail of the ORB.

The Data Streams shall test for possible shared or null values and place appropriate indirections or null encodings (even when used from the custom streaming policy).

There are no explicit operations for creating the streams. It is assumed that the ORB implicitly acts as a factory. In a sense they are always available.

For **write_fixed**, the **fixed_value** parameter must be an "any" containing a fixed value. If the "any" passed in does not contain a fixed value, then a BadFixedValue exception is raised with the offset field set to 0.

For **write_fixed_array**, the elements of the **seq** parameter that are specified by the offset and length parameters must be a sequence of "any"s each of which contains a fixed value. If any of these "any"s do not contain a fixed value, or if any of them contain a fixed value whose **digits** and **scale** (as specified by the **TypeCode** in the "any") differ from those of the first of these "any"s (as specified by its **TypeCode**), then a BadFixedValue exception is raised with the offset field set to a zero-origin ordinal number indicating the position of the first incorrect "any" within the subsequence of fixed values written to the stream.

For both **write_fixed** and **write_fixed_array**, the **TypeCode** within each "any" being written specifies the **digits** and **scale** to be used to write the fixed value contained in the "any." The **TypeCode** itself is not written to the **DataOutputStream**.

The **read_fixed** operation returns an "any" containing the fixed value that was read from the **DataInputStream**. The digits and scale in the **TypeCode** of the returned "any" are set to the **digits** and **scale** parameters passed to **read_fixed**. If the fixed value read from the **DataInputStream** is incompatible with the **digits** and **scale** parameters passed to **read_fixed**, then a BadFixedValue exception is raised with the offset field set to 0.

The **read_fixed_array** operation sets the elements of the **seq** parameter that are specified by the **offset** and **length** parameters. These elements are set to "any"s with **TypeCodes** specifying a fixed value whose **digits** and **scale** are the same as the **digits** and **scale** parameters, and fixed values that were read from the **DataInputStream**. The previous contents of these "any"s, including their **TypeCodes**, are destroyed by the **read_fixed_array** operation. Other "any"s in

the **seq** parameter (if any) are left unchanged. No **TypeCode** information is read from the **DataInputStream**. If any of the fixed values read from the **DataInputStream** are incompatible with the **digits** and **scale** parameters, then a BadFixedValue exception is raised with the **offset** field set to a zero-origin ordinal number indicating the position of the first incorrect “any” within the subsequence of fixed values read from the stream.

The stream representation of a fixed value is considered incompatible if its **digit** and **scale** values do not match the **digits** and **scale** values being used to read it from the stream.

6.6 Access to the Sending Context Run Time

There are two cases where a receiving context might want to access the run time environment of the sending context:

- To attempt the downloading of some missing implementation for the value.
- To access some meta information about the version of the value just received.

In order to provide that kind of service a call back object interface is defined. It may optionally be supported by the sending context (it can be seen as a service). If such a callback object is supported, its IOR may be added to an optional service context in the GIOP header passed from the sending context to the receiving context.

A service context tagged with the ServiceID **SendingContextRunTime** (see *Part 2 of this specification*) contains an encapsulation of the IOR for a **SendingContext::RunTime** object. Because ORBs are always free to skip a service context they don’t understand, this addition does not impact IIOP interoperability.

```
module SendingContext {
    interface RunTime {}; // so that we can provide more
                          // sending context run time
                          // services in the future

    interface CodeBase: RunTime {
        typedef string URL; // blank-separated list of one or more URLs
        typedef sequence<URL> URLSeq;
        typedef sequence
            <CORBA::ValueDef::FullValueDescription> ValueDescSeq;

        // Operation to obtain the IR from the sending context
        CORBA::Repository get_ir();

        // Operations to obtain a location of the implementation code
        URL implementation(in CORBA::RepositoryId x);
        URLSeq implementations(in CORBA::RepositoryIdSeq x);

        // Operations to obtain complete meta information about a Value
        // This is just a performance optimization the IR can provide
        // the same information
        CORBA::FullValueDescription meta(in CORBA::RepositoryId x);
        ValueDescSeq metas(in CORBA::RepositoryIdSeq x);

        // To obtain a type graph for a value type
        // same comment as before the IR can provide similar
```

```
        // information
        CORBA::RepositoryIdSeq bases(in CORBA::RepositoryId x);
    };
};
```

Supporting the **CodeBase** interface for a given ORB run time is an issue of quality of service. The point here is that if the sending context does not support a **CodeBase,** then the receiving context will simply raise an exception with which the sending context had to be prepared to deal. There will always be cases where a receiving context will get a value type and won't be able to interpret it because:

- It can't get a legal implementation for it (even if it knows where it is, possibly due to security and/or resource access issues).
- Its local version is so radically different that it cannot make sense out of the piece of state being provided.

These two failure modes will be represented by the CORBA system exception NO_IMPLEMENT with identified minor codes, for a missing local value implementation and for incompatible versions.

Under certain conditions it is possible that when several values of the same CORBA type (same repository id) are sent in either a request or reply, that the reality is that they have distinct implementations. In this case, in addition to the codebase URL(s) sent in the service context, each value that has a different codebase may have codebase URL(s) associated with it. This is encoded by using a different tag to encode the value on the wire.

The sending context does not need to resend the same value for this service context on subsequent requests over the same underlying connection. Resending a different value for this service context is only necessary if the callback object reference in use is changed by the sending context within the lifetime of the underlying connection.

7 Abstract Interface Semantics

7.1 Overview

In many cases it may be useful to defer the determination of whether an object is passed by reference or by value until runtime. An IDL abstract interface provides this capability. See *Example on page 94* for an example of when this might be useful.

7.2 Semantics of Abstract Interfaces

Abstract interfaces differ from regular IDL interfaces in the following ways:

1. When used in an operation signature, they do not determine whether actual parameters are passed as an object reference or by value. Instead, the type of the actual parameter (regular interface or value) is used to make this determination using the following rules:
   - The actual parameter is passed as an object reference if it is a regular interface type (or a subtype of a regular interface type), and that regular interface type is a subtype of the signature abstract interface type, and the object is already registered with the ORB/OA.
   - The actual parameter is passed as a value if it cannot be passed as an object reference but can be passed as a value. Otherwise, a BAD_PARAM exception is raised.
2. Abstract interfaces do not implicitly inherit from **CORBA::Object.** This is because they can represent either value types or CORBA object references, and value types do not necessarily support the object reference operations. If an IDL abstract interface type can be successfully narrowed to an object reference type (a regular IDL interface), then the **CORBA::Object** operations can be invoked on the narrowed object reference.
3. Abstract interfaces implicitly inherit from **CORBA::AbstractBase**. This type is defined as native. It is the responsibility of each language mapping to specify the actual programming language type that is used for this type.

   ```
   module CORBA {
   // IDL
       native AbstractBase;
   };
   ```
4. Abstract interfaces do not imply copy semantics for value types passed as arguments to their operations. This is because their operations may be either CORBA invocations (for abstract interfaces that represent CORBA object references) or local programming language calls (for abstract interfaces that represent CORBA value types). See Operations on page 80 and Parameter Passing on page 81 for details of these differences.
5. Special inheritance rules that apply to abstract interfaces are described in Abstract Interface on page 25.
6. See the General Inter-orb Protocol clause in the CORBA 3.2 specification - for special consideration when transmitting an abstract interface using GIOP.

In other respects, abstract interfaces are identical to regular IDL interfaces. For example, consider the following operation **m1()** in abstract interface **foo**.

```
abstract interface foo {
```

```
        void m1(in AnInterfaceType x, in AnAbstractInterfaceType y,
            in AValueType z);
};
```

x's are always passed by reference.

z's are passed as:

- copied values if **foo** refers to an ordinary interface.
- non-copied values if **foo** refers to a value type.

y's are passed as:

- reference if their concrete type is an ordinary interface subtype of **AnAbstractInterfaceType** (registered with the ORB), no matter what **foo'**s concrete type is.
- copied values if their concrete type is value and **foo**'s concrete type is ordinary interface.
- non-copied values if their concrete type is value and **foo**'s concrete type is value.

7.3 Usage Guidelines

Abstract interfaces are intended for situations where it cannot be known at compile time whether an object reference or a value will be passed. In other cases, a regular interface or value type should be used. Abstract interfaces are not intended to replace regular CORBA interfaces in situations where there is no clear need to provide runtime flexibility to pass either an object reference or a value. If reference semantics are intended, regular interfaces should be used.

7.4 Example

For example, in a business application it is extremely common to need to display a list of objects of a given type, with some identifying attribute like account number and a translated text description such as "Savings Account." A developer might define an interface such as **Describable** whose methods provide this information, and implement this interface on a wide range of types. This allows the method that displays items to take an argument of type **Describable** and query it for the necessary information. The **Describable** objects passed in to the **display** method may be either CORBA interface types (passed in as object references) or CORBA value types (passed in by value).

In this example, **Describable** is used as a polymorphic abstract type. No instances of type **Describable** exist, but many different instances have interfaces that support the **Describable** type abstraction. In C++, **Describable** would be an abstract base class; in Java, an interface. In statically typed languages, the compiler can check that the actual parameter type passed by callers of **display** is a valid subtype of **Describable** and therefore supports the methods defined by **Describable**. The **display** method can simply invoke the methods of **Describable** on the objects that it receives, without concern for any details of their implementation.

Describable could not be declared as a regular IDL interface. This is because arguments of declared interface type are always passed as object references (see Parameter Passing on page 81) and we also want the **display** method to be able to accept value type objects that can only be passed by value. Similarly we cannot define **Describable** as a value type because then the **display** method would not be able to accept actual parameter objects that only support passing as an object reference. Abstract interfaces are needed to cover such cases.

The **Describable** abstract interface could be defined and used by the following IDL:

```
abstract interface Describable {
        string get_description();
};

interface Example {
        void display (in Describable anObject);
};

interface Account : Describable {// passed by reference
   // add Account methods here
};

valuetype Currency supports Describable {// passed by value
   // add Currency methods here
};
```

If **Describable** was defined as a regular interface instead of an abstract interface, then it would not be possible to pass a **Currency** value to the display method, even though the **Currency** IDL type supports the **Describable** interface.

7.5 Security Considerations

Security considerations for abstract interfaces are similar to those for regular interfaces and values (see Security Considerations on page 84). This is because an abstract interface formal parameter type allows either a regular interface (IOR) or a value to be passed. Likewise, an operation defined in an abstract interface can be implemented by either a regular interface (with “normal” security considerations) or by a value type (in which case it is a local call, not mediated by the ORB). The security implication of making the choice between these alternatives a runtime determination is that the programmer must ensure that for both alternatives, no security violations can occur. For example, a technique similar to that described in “Passing Values to Trusted Domains” could be used to avoid inadvertently passing values outside a domain of trust.

7.5.1 Passing Values to Trusted Domains

When a server passes an object reference, it can be sure that access control policies will apply to any attempt to access anything through that object reference. When the underlying object is passed as a value, the granularity and level/ semantics of access control are different. In the “by value” case, all the data for the object is passed, and method invocations on the passed object are local calls that are not mediated by the ORB. Whether the server wants to use the (potentially more permissive) pass by value access control or not could depend on the security domain, which is receiving the said object or object reference.

Consider the case where the server S has an object O that it is willing to pass only in the form of an object reference Or' to a domain Du that it does not trust, but is willing to pass the object by value Ow to another domain Ot that it trusts.

This flexibility is not possible without abstract interfaces. Signatures would have to be written to either always pass references or always pass values, irrespective of the level of trust of the invocation target domain. However, abstract interfaces provide the necessary flexibility. The formal parameter type **MyType** can be declared as an abstract interface and the method invocation can be coded along the lines of

```
myExample->foo(security_check(myExample,mydata));
```

where the **`security_check`** function determines the level of trust of **`myExample`**'s domain and returns a regular interface subtype of **`MyType`** for untrusted domains and a value subtype of **`MyType`** for trusted domains. The rules for abstract interfaces will then pass the correct thing in both these cases.

Annex A
Legal Information

(informative)

A.1 Copyright Information

A.2 Use Of Specification - Terms, Conditions & Notices

A.3 Licenses

International Standard, and to use, copy, and distribute this International Standard as provided under the Copyright Act; provided that: (1) both the copyright notice identified above and this permission notice appear on any copies of this International Standard; (2) the use of the specifications is for informational purposes and will not be copied or posted on any network computer or broadcast in any media and will not be otherwise resold or transferred for commercial purposes; and (3) no modifications are made to this International Standard. This limited permission automatically terminates without notice if you breach any of these terms or conditions. Upon termination, you will destroy immediately any copies of the specifications in your possession or control.

A.4 Patents

The attention of adopters is directed to the possibility that compliance with or adoption of OMG specifications may require use of an invention covered by patent rights. OMG shall not be responsible for identifying patents for which a license may be required by any OMG specification, or for conducting legal inquiries into the legal validity or scope of those patents that are brought to its attention. OMG specifications are prospective and advisory only. Prospective users are responsible for protecting themselves against liability for infringement of patents.

A.5 General Use Restrictions

Any unauthorized use of this International Standard may violate copyright laws, trademark laws, and communications regulations and statutes. This document contains information which is protected by copyright. All Rights Reserved. No part of this work covered by copyright herein may be reproduced or used in any form or by any means--graphic, electronic, or mechanical, including photocopying, recording, taping, or information storage and retrieval systems--without permission of the copyright owner.

A.6 Disclaimer Of Warranty

WHILE THIS PUBLICATION IS BELIEVED TO BE ACCURATE, IT IS PROVIDED "AS IS" AND MAY CONTAIN ERRORS OR MISPRINTS. THE OBJECT MANAGEMENT GROUP AND THE COMPANIES LISTED ABOVE MAKE NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS PUBLICATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTY OF TITLE OR OWNERSHIP, IMPLIED WARRANTY OF MERCHANTABILITY OR WARRANTY OF FITNESS FOR A PARTICULAR PURPOSE OR USE.

IN NO EVENT SHALL THE OBJECT MANAGEMENT GROUP OR ANY OF THE COMPANIES LISTED ABOVE BE LIABLE FOR ERRORS CONTAINED HEREIN OR FOR DIRECT, INDIRECT, INCIDENTAL, SPECIAL, CONSEQUENTIAL, RELIANCE OR COVER DAMAGES, INCLUDING LOSS OF PROFITS, REVENUE, DATA OR USE, INCURRED BY ANY USER OR ANY THIRD PARTY IN CONNECTION WITH THE FURNISHING, PERFORMANCE, OR USE OF THIS MATERIAL, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

The entire risk as to the quality and performance of software developed using this International Standard is borne by you. This disclaimer of warranty constitutes an essential part of the license granted to you to use this International Standard.

A.7 Restricted Rights Legend

Use, duplication or disclosure by the U.S. Government is subject to the restrictions set forth in subparagraph (c) (1) (ii) of The Rights in Technical Data and Computer Software Clause at DFARS 252.227-7013 or in subparagraph (c)(1) and (2) of the Commercial Computer Software - Restricted Rights clauses at 48 C.F.R. 52.227-19 or as specified in 48 C.F.R.

227-7202-2 of the DoD F.A.R. Supplement and its successors, or as specified in 48 C.F.R. 12.212 of the Federal Acquisition Regulations and its successors, as applicable. The specification copyright owners are as indicated above and may be contacted through the Object Management Group, 109 Highland Avenue, Needham, MA 02494, U.S.A.

A.8 Trademarks

MDA®, Model Driven Architecture®, UML®, UML Cube logo®, OMG Logo®, CORBA® and XMI® are registered trademarks of the Object Management Group, Inc., and Object Management Group™, OMG™ , Unified Modeling Language™, Model Driven Architecture Logo™, Model Driven Architecture Diagram™, CORBA logos™, XMI Logo™, CWM™, CWM Logo™, IIOP™ , MOF™ and OMG Interface Definition Language (IDL)™ , and Systems Modeling Language (SysML™) are trademarks of the Object Management Group. All other products or company names mentioned are used for identification purposes only, and may be trademarks of their respective owners.

A.9 Compliance

The copyright holders listed above acknowledge that the Object Management Group (acting itself or through its designees) is and shall at all times be the sole entity that may authorize developers, suppliers and sellers of computer software to use certification marks, trademarks or other special designations to indicate compliance with these materials.

Software developed under the terms of this license may claim compliance or conformance with this International Standard if and only if the software compliance is of a nature fully matching the applicable compliance points as stated in the specification. Software developed only partially matching the applicable compliance points may claim only that the software was based on this International Standard, but may not claim compliance or conformance with this International Standard. In the event that testing suites are implemented or approved by Object Management Group, Inc., software developed using this International Standard may claim compliance or conformance with the specification only if the software satisfactorily completes the testing suites.