INTERNATIONAL STANDARD

ISO/IEC 14750

First edition
1999-03-15

Information technology — Open Distributed Processing — Interface Definition Language

Technologies de l'information — Traitement distribué ouvert — Langage de définition d'interface

Reference number
ISO/IEC 14750:1999(E)

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS

Not for Resale

© ISO/IEC 1999

All rights reserved. Unless otherwise specified, no part of this publication may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying and microfilm, without permission in writing from the publisher.

ISO/IEC Copyright Office • Case postale 56 • CH-1211 Genève 20 • Switzerland

Printed in Switzerland

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS

Not for Resale

Contents

		Page
1	Scope	1
2	Normative references	1
	2.1 Identical Recommendations | International Standards	1
3	Definitions	1
4	ODP IDL syntax and semantics	1
	4.1 Lexical conventions	2
	4.2 Preprocessing	7
	4.3 ODP IDL grammar	8
	4.4 ODP IDL specification	12
	4.5 Inheritance	13
	4.6 Constant declaration	15
	4.7 Type declaration	17
	4.8 Typecodes and Principals	22
	4.9 Exception declaration	22
	4.10 Operation declaration	23
	4.11 Attribute declaration	25
	4.12 CORBA module	25
	4.13 Names and scoping	25
	4.14 Differences from C++	27
Annex A – Reserved standard exceptions		28
	A.1 Object Non-Existence	29
	A.2 Transaction exceptions	29
Annex B – Typecode encoding in the CORBA specification		30

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work.

International Standards are drafted in accordance with the rules given in the ISO/IEC Directives, Part 3.

In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1. Draft International Standards adopted by the joint technical committee are circulated to national bodies for voting. Publication as an International Standard requires approval by at least 75 % of the national bodies casting a vote.

International Standard ISO/IEC 14750 was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, Subcommittee SC 33, *Distributed application services*, in collaboration with ITU-T. The identical text is published as ITU-T Recommendation X.920.

Annex A forms a normative part of this International Standard. Annex B is for information only.

--`,,```,,,,````-`-`,,`,,`,`,,`---
Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Introduction

The rapid growth of distributed processing has led to a need for a coordinating framework for the standardization of Open Distributed Processing (ODP). The Reference Model of Open Distributed Processing (RM-ODP) provides such a framework. It defines an architecture within which support of distribution, interoperability and portability can be integrated.

One of the components of the architecture (described in RM-ODP Part 3: Architecture) (see ITU-T Rec. X.903 | ISO/IEC 10746-3) is a language that is suitable for describing the signature of computational operation interfaces. This Recommendation | International Standard contains such an Interface Definition Language, called ODP-IDL.

NOTE – This Recommendation | International Standard is technically aligned with the CORBA Interface Definition Language specification.

Annex A is normative and provides a standard set of exceptions for a particular ODP distribution infrastructure.

Annex B is informative and provides the CORBA encoding of a type called TypeCode representing type descriptions.

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

INTERNATIONAL STANDARD

ITU-T RECOMMENDATION

INFORMATION TECHNOLOGY – OPEN DISTRIBUTED PROCESSING – INTERFACE DEFINITION LANGUAGE

1 Scope

This Recommendation | International Standard is intended to provide the ODP Reference Model (see ITU-T Rec. X.902 | ISO/IEC 10746-2 and ITU-T Rec. X.903 | ISO/IEC 10746-3) with a language and environment neutral notation to describe computational operation interface signatures. Use of this notation does not imply use of specific supporting mechanisms and protocols.

2 Normative references

The following Recommendations and International Standards contain provisions which, through reference in this text, constitute provisions of this Recommendation | International Standard. At the time of publication, the editions indicated were valid. All Recommendations and Standards are subject to revision, and parties to agreements based on this Recommendation | International Standard are encouraged to investigate the possibility of applying the most recent edition of the Recommendations and Standards listed below. Members of IEC and ISO maintain registers of currently valid International Standards. The Telecommunication Standardization Bureau of the ITU maintains a list of currently valid ITU-T Recommendations.

2.1 Identical Recommendations | International Standards

- 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.*

2.2 Additional references

- ISO/IEC 646:1991, *Information technology – ISO 7-bit coded character set for information interchange.*
- ISO/IEC 8859-1:1998, *Information technology – 8-bit single-byte coded graphic character sets – Part 1: Latin alphabet No. 1.*

3 Definitions

For the purposes of this Recommendation | International Standard, the following definitions apply.

This Recommendation | International Standard makes use of the following terms defined in ITU-T Rec. X.902 | ISO/IEC 10746-2:

- object;
- interface;
- interface signature.

This Recommendation | International Standard makes use of the following terms defined in ITU-T Rec. X.903 | ISO/IEC 10746-3:

- operation.

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

4 ODP IDL syntax and semantics

ODP IDL (the Interface Definition Language) is the language used to describe the interface signatures for interfaces that client objects call and object implementations provide. An interface definition written in ODP IDL completely defines the interface signature and fully specifies each operation's parameters.

An ODP 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 the preprocessing is a sequence of tokens. Such a sequence of tokens, that is, a file after preprocessing, is called a translation unit.

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

The description of ODP IDL's lexical conventions is presented in 4.1. A description of ODP IDL preprocessing is presented in 4.2. The scope rules for identifiers in an ODP IDL specification are described in 4.13 on.

The ODP IDL grammar is a subset of ISO/IEC C++ with additional constructs to support the operation invocation mechanism. ODP IDL is a descriptive language; it supports C++ syntax for constant, type, and operation declarations; it does not include any algorithmic structures or variables. The grammar is presented in 4.3.

This clause describes ODP IDL semantics and gives the syntax for ODP IDL grammatical constructs. The description of ODP IDL grammar uses a syntax notation that is similar to Extended Backus-Naur format (EBNF). Table 1 lists the symbols used in this format and their meaning.

Table 1 – ODP IDL EBNF format

Symbol	Meaning
::=	Is defined to be
|	Alternatively
<text>	Non-terminal
"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 synctactic unit.
[]	The enclosed syntactic unit is optional – may occur zero or more times.

4.1 Lexical conventions

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

ODP IDL uses the ISO/IEC Latin-1 (ISO/IEC 8859-1) character set. This character set is divided into alphabetic characters (letters), digits, graphic characters, the space (blank) character and formatting characters. Table 2 shows the ODP IDL alphabetic characters; upper- and lower-case equivalencies are paired. Table 3 shows the digits and Table 4 shows the graphic characters.

The formatting characters are shown in Table 5.

---

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

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

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Table 2 – The 114 alphabetic characters (letters)

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

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Table 3 – Decimal digits

0 1 2 3 4 5 6 7 8 9

Table 4 – The 65 graphic characters

Char	Description	Char	Description
!	Exclamation point	¡	Inverted exclamation mark
"	Double quote	¢	Cent sign
#	Number sign	£	Pound sign
$	Dollar sign	¤	Currency sign
%	Percent sign	¥	Yen sign
&	Ampersand	¦	Broken bar
'	Apostrophe	§	Section/paragraph sign
(	Left parenthesis	¨	Diaeresis
)	Rigth parenthesis	©	Copyright sign
*	Asterisk	ª	Feminine ordinal indicator
+	Plus sign	«	Left angle quotation mark
,	Comma	¬	Not sign
-	Hyphen, minus sign	–	Soft hyphen
.	Period, full stop	®	Registered trade mark sign
/	Solidus	¯	Macron
:	Colon	°	Ring above, degree sign
;	Semicolon	±	Plus-minus sign
<	Less-than sign	2	Superscript two
=	Equals sign	3	Superscript three
>	Greater-than sign	´	Acute
?	Question mark	µ	Micro
@	Commercial at	¶	Pilcrow
[	Left square bracket	·	Middle dot
\	Reverse solidus	¸	Cedilla
]	Right square bracket	1	Superscript one
^	Circumflex	º	Masculine ordinal indicator
_	Low line, underscore	»	Right angle quotation mark
`	Grave	¼	Vulgar fraction 1/4
{	Left curly bracket	½	Vulgar fraction 1/2
|	Vertical line	¾	Vulgar fraction 3/4
}	Right curly bracket	¿	Inverted question mark
~	Tilde	x	Multiplication sign
		÷	Division sign

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Table 5 – The Formatting Characters

Description	Abbreviation	ISO/IEC 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

4.1.1 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.

4.1.2 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.

4.1.3 Identifiers

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

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

When comparing two identifiers to see if they collide:

- Upper- and lower-case letters are treated as the same letter. Table 2 defines the equivalence mapping of upper- and lower-case letters.
- The comparison does *not* take into account equivalences between digraphs and pairs of letters (e.g. "æ" and "ae" are not considered equivalent) or equivalences between accented and non-accented letters (e.g. "à" and "a" are not considered equivalent).
- All characters are significant.

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

4.1.4 Keywords

The identifiers listed in Table 6 reserved for use as keywords, and may not be used otherwise. The keyword `Object` in ODP IDL is used to represent an interface type whereas the keyword `interface` is used to indicate the start of an interface declaration in an interface signature template. The keyword "`Object`" can be used as a type specifier. The keyword `attribute` defines a method giving access to a portion of the state of an object. An attribute definition is logically equivalent to declaring a pair of accessor methods; one to retrieve the value of the attribute and one to set the value of the attribute.

The keyword Exception is used to represent the ODP concept of unsuccessful named termination, the exception name being the termination name.

Keywords obey the rules for identifiers (see 1.3) and must be written exactly as shown in the above list. For example, "`boolean`" is correct; "`Boolean`" is not. ODP IDL specifications use the characters shown in Table 7 as punctuation.

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

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Table 6 – Keywords

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

Table 7 – Punctuation tokens

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

Table 8 – Preprocessor tokens

#	##	!	||	&&	include	pragma	define

4.1.5 Literals

Wide character and wide string literals are specified exactly like character and string literals. All character and string literals, both wide and non-wide, may only be specified (portably) using the characters found in the ISO/IEC 8859-1 character set. Note that these extensions for international characters only affect the specification of literals in the ODP IDL and not the rest of ODP IDL source files. That is, the interface names, operation names, type names, etc., will continue to be limited to the ISO/IEC 8859-1 character set.

Literals of the new integer and floating-point types are specified as described in this subclause (*Integer, Literals* and *Floating-Point Literals*).

4.1.5.1 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 preceeded 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.

4.1.5.2 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 characterliteral is the numerical value of the character as defined in the ISO/IEC Latin-1 (ISO/IEC 8859-1) character set standard (see Tables 2, 3, and 4). The value of a null is 0. The value of a formatting character literal is the numerical value of the character as defined in the ISO/IEC 646 standard (see Table 5). The meaning of all other characters is implementation-dependent.

Non-graphic characters must be represented using escape sequences as defined below in Table 9. Note that escape sequences must be used to represent singlequote and backlash characters in character literals.

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

The escape \000 consists of the backlash 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 backlash followed by x followed by one or two hexadecimal digits that are taken to specify the value of the desired character. A sequence of octal or hexadecimal digits is terminated by the first character that is not an octal digit or a hexadecimal digit respectively. The value of a character constant is implementation dependent if it exceeds that of the largest character.

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Table 9 – Escape sequences

Description	Escape sequence
Newline	\n
Horizontal tab	\t
Vertical tab	\v
Backspace	\b
Carriage return	\r
Form feed	\f
Alert	\a
Backslash	\\
Question mark	\?
Single quote	\’
Double quote	\"
Octal number	\ooo
Hexadecimal number	\xhh

4.1.5.3 Floating-point literals

A floating-point 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.

4.1.5.4 Fixed-point literal

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.

4.1.5.5 String literals

A string literal is a sequence of characters (as defined in 4.1.5.2) surrounded by double quotes, as in "...".

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

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

The size of a string literal is the number of character literals enclosed by the quotes, after concatenation. The size of the literal is associated with the literal.

4.2 Preprocessing

A preprocessing notation can be used as a module notation, in order to organize specifications and to be able to refer to parts of a specification in a given specification. Therefore, the source file inclusion #include must be understood as a generic way of including a given module of specification and is not linked with any particular filing system.

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

Lines beginning with # (also called "directives") communicate with this preprocessor. White space may appear before the #. These lines have syntax independent of the rest of the ODP IDL; they may appear anywhere and have effects that last (independent of the ODP IDL scoping rules) until the end of the translation unit. The textual location of ODP 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.

--`,,```,,,,````-`-`,,`,,`,`,,`---
Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

A preprocessing token is an ODP IDL token (see 4.1.1), 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 ODP IDL specifications. Text in files included with a `#include` directive is treated as if it appeared in the including file. A complete description of the preprocessing facilities may be found in the *ANSI/ISO C++ Standard*.

4.3 ODP IDL grammar

```
(1)   <specification>          ::=  <definition>+
(2)   <definition>             ::=  <type_dcl>";"
                               |    <const_dcl>";"
                               |    <except_dcl>";"
                               |    <interface>";"
                               |    <module>";"
(3)   <module>                 ::=  "module"  <identifier> "{"  <definition>+ "}"
(4)   <interface>              ::=  <interface_dcl>
                               |    <forward_dcl>
(5)   <interface_dcl>          ::=  <interface_header> "{" <interface_body> "}"
(6)   <forward_dcl>            ::=  "interface" <identifier>
(7)   <interface_header>       ::=  "interface" <identifier> [<inheritance_spec> ]
(8)   <interface_body>         ::=  <export>*
(9)   <export>                 ::=  <type_dcl> ";"
                               |    <const_dcl> ";"
                               |    <except_dcl> ";"
                               |    <attr_dcl> ";"
                               |    <op_dcl> ";"
(10)  <inheritance_spec>       ::=  ":" <scoped_name> { "," <scoped_name> }*
(11)  <scoped_name>            ::=  <identifier>
                               |    "::" <identifier>
                               |    <scoped_name> "::" <identifier>
(12)  <const_dcl>              ::=  "const" <const_type> <identifier> "=" <const_exp>
(13)  <const_type>             ::=  <integer_type>
                               |    <char_type>
                               |    <wide_char_type>
                               |    <boolean_type>
                               |    <floating_pt_type>
                               |    <string_type>
                               |    <wide_string_type>
                               |    <fixed_pt_const_type>
                               |    <scoped_name>
(14)  <const_exp>              ::=  <or_exp>
(15)  <or_exp>                 ::=  <xor_expr>
                               |    <or_expr> "|" <xor_expr>
(16)  <xor_expr>               ::=  <and_expr>
                               |    <xor_expr> "^" <and_expr>
(17)  <and_expr>               ::=  <shift_expr>
                               |    <and_expr> "&" <shift_expr>
```


Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
(18)   <shift_expr>            ::=  <add_expr>
                              |    <shift_expr> ">>" <add_expr>
                              |    <shift_expr> "<<" <add_expr>
(19)   <add_expr>              ::=  <mult_expr>
                              |    <add_expr> "+" <mult_expr>
                              |    <add_expr> "-" <mult_expr>
(20)   <mult_expr>             ::=  <unary_expr>
                              |    <mult_expr> "*" <unary-expr>
                              |    <mult_expr> "/" <unary_expr>
                              |    <mult_expr> "%" <unary_expr>
(21)   <unary_expr>            ::=  <unary_operator> <primary_expr>
                              |    <primary_expr>
(22)   <unary_operator>        ::=  "-"
                              |    "+"
                              |    "~"
(23)   <primary_expr>          ::=  <scoped_name>
                              |    <literal>
                              |    "(" <const_exp> ")"
(24)   <literal>               ::=  <integer_literal>
                              |    <string_literal>
                              |    <wide_string_literal>
                              |    <character_literal>
                              |    <wide_character_literal>
                              |    <fixed_pt_literal>
                              |    <floating_pt_literal>
                              |    <boolean_literal>
(25)   <boolean_literal>       ::=  "TRUE"
                              |    "FALSE"
(26)   <positive_int_const>    ::=  <const_exp>
(27)   <type_dcl>              ::=  "typedef" <type_declarator>
                              |    <struct_type>
                              |    <union_type>
                              |    <enum_type>
(28)   <type_declarator>       ::=  <type_spec> <declarators>
(29)   <type_spec>             ::=  <simple_type_spec>
                              |    <constr_type_spec>
(30)   <simple_type_spec>      ::=  <base_type_spec>
                              |    <template_type_spec>
                              |    <scoped_name>
(31)   <base_type_spec>        ::=  <floating_pt_type>
                              |    <integer_type>
                              |    <char_type>
                              |    <wide_char_type>
                              |    <boolean_type>
```

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
                                |    <octet_type>
                                |    <any_type>
                                |    <object_type>
(31a)  <object_type>            ::=  "Object"
(32)   <template_type_spec>     ::=  <sequence_type>
                                |    <string_type>
                                |    <wide_string_type>
                                |    <fixed_pt_type>
(33)   <constr_type_spec>       ::=  <struct_type>
                                |    <union_type>
                                |    <enum_type>
(34)   <declarators>            ::=  <declarator> { "," <declarator> } *
(35)   <declarator>             ::=  <simple_declarator>
                                |    <complex_declarator>
(36)   <simple_declarator>      ::=  <identifier>
(37)   <complex_declarator>     ::=  <array_declarator>
(38)   <floating_pt_type>       ::=  "float"
                                |    "double"
                                |    "long" "double"
(39)   <integer_type>           ::=  <signed_int>
                                |    <unsigned_int>
(40)   <signed_int>             ::=  <signed_long_int>
                                |    <signed_short_int>
                                |    <signed_longlong_int>
(40a)  <signed_longlong_int>    ::=  "long" "long"
(41)   <signed_long_int>        ::=  "long"
(42)   <signed_short_int>       ::=  "short"
(43)   <unsigned_int>           ::=  <unsigned_long_int>
                                |    <unsigned_short_int>
                                |    <unsigned_longlong_int>
(43a)  <unsigned_longlong_int>  ::= "unsigned" "long" "long"
(44)   <unsigned_long_int>      ::=  "unsigned" "long"
(45)   <unsigned_short_int>     ::=  "unsigned" "short"
(46)   <char_type>              ::=  "char"
(46a)  <wide_char_type>         ::=  "wchar"
(47)   <boolean_type>           ::=  "boolean"
(48)   <octet_type>             ::=  "octet"
(49)   <any_type>               ::=  "any"
(50)   <struct_type>            ::=  "struct" <identifier> "{" <member_list> "}"
(51)   <member_list>            ::=  <member>+
(52)   <member>                 ::=  <type_spec> <declarators> ";"
(53)   <union_type>             ::=  "union"<identifier> "switch"
                                          "(" <switch_type_spec> ")" "{" <switch_body> "}"
(54)   <switch_type_spec>       ::=  <integer_type>
                                |    <char_type>
                                |    <boolean_type>
```


Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
                                |    <enum_type>
                                |    <scoped_name>
(55)   <switch_body>            ::=  <case>+
(56)   <case>                   ::=  <case_label>+ <element_spec> ";"
(57)   <case_label>             ::=  "case" <const_exp> ":"
                                |    "default" ":"
(58)   <element_spec>           ::=  <type_spec> <declarator>
(59)   <enum_type>              ::=  "enum" <identifier> "{" <enumerator>
                                          {"," <enumerator> } * "}"
(60    <enumerator>             ::=  <identifier>
(61)   <sequence_type>          ::=  "sequence" "<" <simple_type_spec> ","
                                          <positive_int_const> ">"
                                |    "sequence" "<" <simple_type_spec> ">"
(62)   <string_type>            ::=  "string" "<" <positive_int_const> ">
                                |    "string"
(62a)  <wide_string_type>       ::=  "wstring" "<" <positive_int_const>
                                          ">" | "wstring"
(63)   <array_declarator>       ::=  <identifier> <fixed_array_size>+
(64)   <fixed_array_size>       ::=  "[" <positive_int_const> "]"
(65)   <attr_dcl>               ::=  [ "readonly" ] "attribute"<param_type_spec>
                                          <simple_declarator>
                                          { "," <simple_declarator> }*
(66)   <except_dcl>             ::=  "exception" <identifier> "{" <member>*"}"
(67)   <op_dcl>                 ::=  [ <op_attribute> ] <op_type_spec>
                                          <identifier> <parameter_dcls>
                                          [ <raises_expr> ] [ <context_expr> ]
(68)   <op_attribute>           ::=  "oneway"
(69)   <op_type_spec>           ::=  <param_type_spec>
                                |    "void"
(70)   <parameter_dcls>         ::=  "(" <param_dcl> { "," <param_dcl> }* ")"
                                |    "(" ")"
(71)   <param_dcl>              ::=  <param_attribute> <param_type_spec>
                                          <simple_declarator>
(72)   <param_attribute>        ::=  "in"
                                |    "out"
                                |    "inout"
(73)   <raises_expr>            ::=  "raises" "(" <scoped_name> { ","<scoped_name> }* ")"
(74)   <context_expr>           ::=  "context "(" <string_literal>
                                          { ","<string_literal> } * ")"
(75)   <param_type_spec>        ::=  <base_type_spec>
                                |    <string_type>
                                |    <fixed_pt_type>
                                |    <wide_string_type>
                                |    <scoped_name>
(76)   <fixed_pt_type>          ::=  "fixed" "<" <positive_int_const>
                                          ","<integer_literal> ">"
(77)   <fixed_pt_const_type>    ::=  "fixed"
```

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

4.4 ODP IDL specification

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

```
<specification>::=      <defintion> +
<definition>            ::=  <type_dcl> ";"
                        |    <const_dcl> ";"
                        |    <except_dcl> ";"
                        |    <interface_dcl> ";"
                        |    <module_dcl> ";"
```

See 4.6, 4.7 and 4.9, respectively, for specifications of `<const_dcl>`, `<type_dcl>`, and `<except_dcl>`.

4.4.1 Module declaration

A module definition satisfies the following syntax:

```
<module>                ::=  "module" <identifier> "{ <definition>+"}"
```

The module construct is used to scope ODP IDL identifiers; see 4.12 for details.

4.4.2 Interface declaration

An interface declaration satisfies the following syntax:

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

4.4.2.1 Interface header

The interface header consists of two elements:

- The interface name – The name must be preceeded by the keyword `interface`, and consists of an identifier that names the interface.
- An optional inheritance specification – The inheritance specification is described in 4.4.2.2.

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 subclauses. The `Object` name is a valid name which allows to pass any interface reference. Since one can only hold references to an interface , the meaning of a parameter or structure member which is an interface type is as a *reference* to an instance of that interface type. Each language binding describes how the programmer must represent such interface references. In particular, it can be used as parameter in an operation description, which allows to pass interface references as parameters.

4.4.2.2 Inheritance specification

The syntax for inheritance is as follows:

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


Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Each <scoped_name> in an <inheritance_spec> must denote a previously defined interface. See 4.5 for the description of inheritance.

4.4.2.3 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 4.6.
- Type declarations, which specify the type definitions that the interface exports; type declaration syntax is described in 4.7.
- Exception declarations, which specify the exception structures that the interface exports; exception declaration syntax is described in 4.9.
- Attribute declarations, which specify the associated attributes exported by the interface; attribute declaration syntax is described in 4.11.
- 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 paramters of an operation, legal exceptions which may be returned as a result of an invocation, and contextual information which may affect method dispatch; operation declaration syntax is described in 4.10.

Empty interfaces (i.e. those that contain no declarations) are permitted.

4.4.2.4 Forward declaration

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

Multiple forward declarations of the same interface name are legal.

4.5 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 additon, unless redefined in the derived interface, the elements of a base interface can be referred to as if they were elements of the derived interface. The name resolution operator ("::") may be used to refer to a base element explicitly; this permits reference to a name that has been redefined in the derived interface.

A derived interface may redefine any of the type, constant, and exception names which have been inherited; the scope rules for such names are described in 4.12.

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

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

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

```
interface A {...}

interface B:A {...}

interface C:A {...}

interface D:B,C {...}
```

The relationships between these interfaces are shown in Figure 1. This "diamond" shape is legal.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

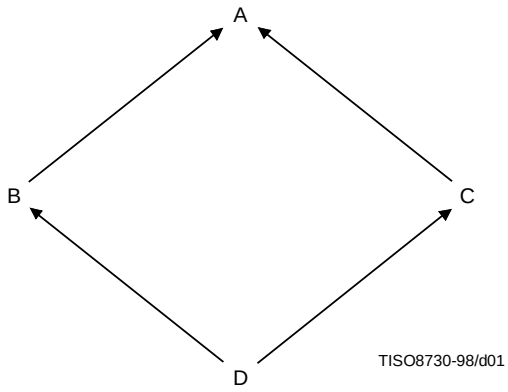


Figure 1 – Legal multiple inheritance example

Reference to base interface elements must be unambiguous. Reference to a base interface element is ambiguous if the expression used refers to a constant type, or exception in more than one base interface. (It is currently illegal to inherit from two interfaces with the same operation or attribute name, or to redefine an operation or attribute name in the derived interface.) Ambiguities can be resolved by qualifying a name with its interface name (i.e. using a <scoped name>).

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

```
const long L=3;

interface A {

         typedef float coord[L]);

      void f (in coord s); // s has three floats };

interface B{

      const long L=4;

};

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

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

```
typedef float coord[3];

void f (in coord s);
```

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

Interface inheritance causes all identifiers in the closure of the inheritance tree to be imported into the current naming scope. A type name, constant name, enumeration value name, or exception name from an enclosing scope can be redefined in the currentscope.

All operations that might apply to a particular object must have unique names. This requirement prohibits redefining an operation name in a derived interface, as well as inheriting two operations with the same name.

NOTE – It is anticipated that future revisions of the language may relax this rule in some way, perhaps allowing overloading or providing some means to distinguish operations with the same name.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

4.6 Constant declaration

4.6.1 Syntax

This subclause describes the syntax for constant declarations.

The syntax for a constant declaration is:

```
<const_dcl>             ::=  "const" <const_type> <identifier> "=" <const_exp>
<const_type>            ::=  <integer_type>
                        |    <char_type>
                        |    <wide_char_type>
                        |    <boolean_type>
                        |    <floating_pt_type>
                        |    <string_type>
                        |    <wide_string_type>
                        |    <fixed_pt_const_type>
                        |    <scoped_name>
<const_exp>             ::=  <or_exp>
<or_exp>                ::=  <xor_expr>
                        |    <or_expr> "|" <xor_expr>
<xor_expr>              ::=  <and_expr>
                        |    <xor_expr> "^" <and_expr>
<and_expr>              ::=  <shift_expr>
                        |    <and_expr> "&" <shift_expr>
<shift_expr>            ::=  <add_expr>
                        |    <shift_expr> ">>" <add_expr>
                        |    <shift_expr> "<<" <add_expr>
<add_expr>              ::=  <mult_expr>
                        |    <add_expr> "+" <mult_expr>
                        |    <add_expr> "-" <mult_expr>
<mult_expr>             ::=  <unary_expr>
                        |    <mult_expr> "*" <unary-expr>
                        |    <mult_expr> "/" <unary_expr>
                        |    <mult_expr> "%" <unrary_expr>
<unary_expr>            ::=  <unary_operator> <primary_expr>
                        |    <primary_expr>
<unary_operator>        ::=  "-"
                        |    "+"
                        |    "~"
<primary_expr>          ::=  <scoped_name>
                        |    <literal>
                        |    "(" <const_exp> ")"
<literal>               ::=  <integer_literal>
                        |    <string_literal>
                        |    <character_literal>
                        |    <wide_character_literal>
```


Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
                              |    <fixed_pt_literal>
                              |    <floating_pt_literal>
                              |    <boolean_literal>
        <boolean_literal>     ::=  "TRUE"
                              |    "FALSE"
        <positive_int_const>::= <const_exp>
```

4.6.2 Semantics

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

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

An integer constant expression is evaluated as unsigned long long unless it contains a negated integer literal or an integer constant with negative value, or other sub-expression with negative value, in which case it is evaluated as a signed long long. The computed value is coerced back to the target type in constant initializers. It is an error if the computed value exceeds the range of the target type, or if any intermediate value exceeds the range of the evaluated-as type (long long or unsigned long long).

All floating point literals are long double, all floating-point constants are coerced to long double, and all floating-point expressions are computed as long doubles. The computed long double value is coerced back to the target type in constant initializers. It is an error if this coercion fails or if any intermediate values (when evaluating the expression) exceed the range of long double.

Unary (+) and binary (* / + –) operators are applicable in floating-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:

```
long long           –(value+1)
unsigned long 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 < 32.

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.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Fixed-point decimal constant expressions are evaluated as follows. A fixed-point literal has the apparent number of total and fractional digits, except that leading and trailing zeroes are factored out, including non-significant zeros before the decimal point. For example, `0123.450d` is considered to be `fixed<5,2>` and `3000.00` is `fixed<1,-3>`. Prefix operators do not affect the precision; a prefix `+` is optional, and does not change the result. The upper bounds on the number of digits and scale of the result of an infix expression, `fixed<d1,s1> op fixed<d2,s2>`, are shown in the following table:

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

A quotient may have an arbitrary number of decimal places, denoted by a scale of `s` inf. The computation proceeds pairwise, with the usual rules for left-to-right association, operator precedence and parentheses. If an individual computation between a pair of fixed-point literals actually generates more than 31 significant digits, then a 31-digit result is retained as follows:

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

Leading and trailing zeroes 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.

4.7 Type declaration

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

```
<type_dcl>              ::=  "typedef" <type_declarator>
                        |    <struct_type>
                        |    <union_type>
                        |    <enum_type
<type_declarator>       ::= <type_spec> <declarators>
```

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

```
<type_spec>             ::=  <simple_type_spec>
                        |    <constr_type_spec>
<simple_type_spec>      ::=  <base_type_spec>
                        |    <template_type_spec>
                        |    <scoped_name>
<base_type_spec>        ::=  <floating_pt_type>
                        |    <integer_type>
                        |    <char_type>
                        |    <wide_char_type>
                        |    <boolean_type>
                        |    <octet_type>
                        |    <any_type>
                        |    <object_type>
```

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
<template_type_spec>    ::=  <sequence_type>
                         |   <string_type>
<constr_type_spec>      ::=  <struct_type>
                         |   <union_type>
                         |   <enum_type>
<declarators>           ::=  <declarator> { "," <declarator> } *
<declarator>            ::=  <simple_declarator>
                         |   <complex_declarator>
<simple_declarator>     ::=  <identifier>
<complex_declarator>    ::=  <array_declarator>
```

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

4.7.1 Basic types

The syntax for the supported basic types is as follows:

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

Each ODP IDL data type is mapped to a native data type via the appropriate language mapping. Conversion errors between ODP IDL data types and the native types to which they are mapped can occur during the performance of an operation invocation.

4.7.1.1 Integer types

ODP IDL supports both signed and unsigned extended-precision integer data types (`long`,`short`, `long long` and `unsigned` integer data types). Type `long` represents the range -2^{31} .. 2^{31} –1, type `unsigned long` represents the range 0 .. 2^{32} –1, type `short` represents the range -2^{15} .. 2^{15} –1, type `unsigned short` represents the range 0 .. 2^{16} –1. Type `long long` represents the range -2^{63} –1, while type `unsigned long long` represents values in the range 0 .. 2^{64} –1.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

4.7.1.2 Fixed type

The `fixed` data type represents a fixed-point decimal number of up to 31 significant digits. The scale factor is normally a non-negative integer less than or equal to the total number of digits (note that constants with effectively negative scale, such as 10 000, are always permitted.). However, some languages and environments may be able to accommodate types that have a negative scale or a scale greater than the number of digits.

4.7.1.3 Floating-point types

ODP 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` type represents an IEEE double-extended floating-point number, which supports an exponent of at least 15 bits in length and a signed fraction of at least 64 bits. The IEEE floating point standard specification (*IEEE Standard for Binary Floating-Point Arithmetic*, ANSI/IEEE Std 754-1985) should be consulted for more information on the precision afforded by these types.

Implementations that do not fully support the value set of the IEEE 754 floating-point standard must completely specify their deviance from the standard.

4.7.1.4 Char type

ODP IDL defines a `char` data type that is an 8-bit quantity which:

1) encodes a single-byte character from any byte-oriented code set; or
2) when used in an array, encodes a multi-byte character from a multi-byte code set.

In other words, an implementation is free to use any code set internally for encoding character data, though conversion to another form may be required for transmission.

By default, the ISO/IEC Latin-1 (ISO/IEC 8859-1) character set standard defines the meaning and representation of all possible graphic characters (i.e. the space, alphabetic, digit and graphic characters defined in Tables 2, 3 and 4). The meaning and representation of the null and formatting characters (see Table 5) is the numerical value of the character as defined in the ASCII (ISO/IEC 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.

4.7.1.5 Wide char type

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

4.7.1.6 Boolean type

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

4.7.1.7 Octet type

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

4.7.1.8 Any type

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

4.7.2 Constructed types

The constructed types are:

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

Although the ODP IDL syntax allows the generation of recursive constructed type specifications, the only recursion permitted for constructed types is through the use of the `sequence` template type.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

For example, the following is legal:

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

See 4.7.3.1 for details of the `sequence` template type.

4.7.2.1 Structures

The structure syntax is:

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

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

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

4.7.2.2 Discriminated unions

The discriminated `union` syntax is:

```
<union_type>            ::=  "union"<identifier> "switch"
                                  "(" <switch_type_spec>"")" "{" <switch_body> "}"
<switch_type_spec>      ::=  <integer_type>
                        |    <char_type>
                        |    <boolean_type>
                        |    <enum_type>
                        |    <scoped_name>
<switch_body>           ::=  <case>+
<case>                  ::=  <case_label>+ <element_spec> ";"
<case_label>            ::=  "case" <const_exp> ":"
                        |    "default" ":"
<element_spec>          ::=  <type_spec> <declarator>
```

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

Case labels must match or be automatically castable to the defined type of the discriminator. The complete set of matching rules are shown in Table 10.

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

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

- if the discriminator value was explicitly listed in a `case` statement, the value of the element associated with that `case` statement;
- if a default `case` label was specified, the value of the element associated with the default `case` label;
- no additional value.

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

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Table 10 – Case label matching

Descriminator type	Matched by
long	Any integer value in the value range of long
long long	Any integer value in the range of long long
short	Any integer value in the value range of short
unsigned long	Any integer value in the value range of unsigned long
unsigned long long	Any integer value in the range of unsigned long long
unsigned short	Any integer value in the value range of unsigned short
char	char
wchar	wchar
boolean	TRUE or FALSE
enum	Any enumerator for the discriminator enumtype

4.7.2.3 Enumerations

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

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

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

4.7.3 Template types

The template types are:

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

4.7.3.1 Sequences

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

The syntax is:

```
<sequence_type>         ::=  "sequence" "<" <simple_type_spec>
                                   ","<positive_int_const> ">"
                         |   "sequence" "<" <simple_type_spec> ">"
```

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

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

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

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.

4.7.3.2 Strings

ODP 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 filed in a structure or union) the length of the string must be set in a language-mapping dependent manner. The syntax is:

```
<string_type>          ::=  "string" "<" <positive_int_const> ">"
                       |    "string"
```

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

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

4.7.3.3 Wide char strings

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

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

4.7.4 Complex declarator

4.7.4.1 Arrays

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

The syntax for arrays is:

```
<array_declarator>     ::= <identifier> <fixed_array_size>+
<fixed_array_size>     ::= "[" <positive_int_const> "]"
```

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

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

4.8 Typecodes and Principals

Typecode and Principal are particular types which can be used to have types as values, in particular as parameters in operations. As such they must be defined by a language binding for a particular implementation. Annex B gives a particular coding for Typecodes in the CORBA specification.

4.9 Exception declaration

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

```
<except_dcl>           ::="exception" <identifier> "{" <member>* "}"
```


Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Each exception is characterized by its ODP 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.

NOTE – This is an architecturally neutral standard, however certain exceptions are reserved and can be found in Annex A.

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.

4.10 Operation declaration

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

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

<op_type_spec>          ::= <param_type_spec>
                            | "void"
```

An operation declaration consists of:

- An optional operation attribute that specifies which invocation semantics the communication system should provide when the operation is invoked. Operation attributes are described in 4.10.1.
- The type of the operation's return result – The type may be any type which can be defined in ODP 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 4.10.2.
- An optional raises expression which indicates which exceptions may be raised as a result of an invocation of this operation. Raises expressions are described in 4.10.3.
- An optional context expression which indicates which elements of the request context may be consulted by the method that implements the operation. Context expressions are described in 4.10.4.

4.10.1 Operation attribute

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

```
<op_attribute>          ::= "oneway"
```

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

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

4.10.2 Parameter declarations

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

```
<parameter_dcls>        ::=  "(" <param_dcl> { "," <param_dcl> }* ")"
                             |   "(" ")"
<param_dcl>             ::=  <param_attribute>
                                  <param_type_spec><simple_declarator>
<param_attribute>       ::=  "in"
                             |   "out"
                             |   "inout"
```

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
<param_type_spec>       ::=  <base_type_spec>
                             |    <string_type>
                             |    <wide_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 `outand inout` parameters are undefined.

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

4.10.3 Raises expressions

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

```
<raises_expr>          ::=  "raises" "(" <scoped_name> { "," <scoped_name> }* ")"
```

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

In addition to any operation-specific exceptions specified in the `raises` expression, there are a standard set of exceptions that may be signalled

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

4.10.4 Context expressions

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

```
<context_expr>         ::=  "context" "(" <string_literal>
                                    {"," <string_literal> }*")"
```

The runtime 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 distribution infrastructure and/or object is free to use information in this *request context* during request resolution and performance.

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

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

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

4.11 Attribute declaration

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

The syntax for `attribute` declaration is:

```
<attr_dcl>              ::= ["readonly"]"attribute"<param_type_spec>
                                <simple_declarator>

                            {","<simple_declarator>}*
```

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

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

      attribute float radius;
      attribute material_t material;
      readonly attribute position_t position;
      ...
};
```

The attribute declarations are equivalent to the following pseudo-specification fragment:

```
float           _get_radius  ();
void            _set_radius  (in float r);
material_t      _get_material();
void   _        set_material (in material_t m);
position_t      _get_position ();
```

The attribute name is subject to ODP IDL's name scoping rules; the access or function names are guaranteed not to collide with any legal operation names specifiable in ODP IDL.

Attribute operations only return errors by means of standard exceptions.

Attributes are inherited. An attribute name cannot be redefined to be a different type. See 4.12 for more information on redefinition constraints and the handling of ambiguity.

4.12 CORBA module

The naming scope CORBA (called the CORBA module) is reserved in the ODP IDL. The definitions within the CORBA module should be used only when writing domain specific interface definitions, and not used in general ODP computational interface descriptions.

4.13 Names and scoping

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

– module;

– interface;

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

- structure;
- union;
- operation;
- exception.

Identifiers for the following kinds of definitions are scoped:

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

An identifier can only be defined once in a scope. However, identifiers can be redefined in nested scopes.

Due to possible restrictions imposed by future language bindings, ODP IDL identifiers are case insensitive, i.e. 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 occurence. (This allows natural mappings to case-sensitive languages.)

Type names defined in a scope are available for immediate use within that scope. In particular, see 4.7.2 on cycles in type definitions.

A name can be used in an unqualified form within a particular scope; it will be resolved by successively searching farther out in enclosing scopes. Once an unqualified name is used in a scope, it cannot be redefined, i.e. if one has used a name defined in an enclosing scope in the current scope, one cannot then redefine a version of the name in the current scope. Such redefinitions yield a compilation error.

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

When a qualified name begins with "::", the resolution process starts with the smallest enclosing naming scope, and locates subsequent identifiers in the qualified name by the rule described in the previous paragraph.

Every ODP 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 ODP 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 identifer 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 ODP IDL definition is the concatenation of the current root, the current scope, a "::", and the <identifier> which is the local name for that definition.

Inheritance produces shadow copies of the inherited identifiers, i.e. it introduces names into the derived interface, but these names are considered to be semantically "the same" as the original definition. Two shadow copies of the same original (as results from the diamond shape of Figure 1) introduce a single name into the derived interface and do not conflict with each other.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

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

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

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

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

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

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

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

interface C:A,B {
      attribute string_t Title; /* AMBIGUOUS */
};
```

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

4.14 Differences from C++

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

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

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Annex A

Reserved standard exceptions

(This annex forms an integral part of this Recommendation | International Standard)

This annex presents a standard set of exceptions defined for an ODP infrastructure. These exception identifiers may be returned as a result of any operation invocation, regardless of the interface specification. Standard exceptions may not be listed in `raises` expressions.

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

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

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

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

```
#define ex_body {unsigned long minor; completion_status completed}


enum completion_status {COMPLETED_YES, COMPLETED_NO, COMPLETED_MAYBE};
enum exception_type {NO_EXCEPTION, USER_EXCEPTION, SYSTEM_EXCEPTION};


exception UNKNOWN                ex_body;    //the unknown exception
exception BAD_PARAM              ex_body;    //an invalid parameter was passed
exception NO_MEMORY              ex_body;    //dynamic memory allocation failure
exception IMP_LIMIT              ex_body;    //violated implementation limit
exception COMM_FAILURE           ex_body;    //communication failure
exception INV_OBJREF             ex_body;    //invalid object reference
exception NO_PERMISSION          ex_body;    //no permission for attempted op.
exception INTERNAL               ex_body;    //ORB internal error
exception MARSHAL                ex_body;    //error marshalling param/result
exception INITIALIZE             ex_body;    //ORB initialization failure
exception NO_IMPLEMENT           ex_body;    //operation implementation unavailable
exception BAD_TYPECODE           ex_body;    //bad typecode
exception BAD_OPERATION          ex_body;    //invalid operation
exception NO_RESOURCES           ex_body;    //insufficient resources for req.
exception NO_RESPONSE            ex_body;    //response to req. not yet available
exception PERSIST_STORE          ex_body;    //persistent storage failure
```


Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
exception BAD_INV_ORDER             ex_body;    //routine invocations out of order
exception TRANSIENT                 ex_body;    //transient failure – reissue request
exception FREE_MEM                  ex_body;    //cannot free memory
exception INV_IDENT                 ex_body;    //invalid identifier syntax
exception INV_FLAG                  ex_body;    //invalid flag was specified
exception INTF_REPOS                ex_body;    //error accessing interface repository
exception BAD_CONTEXT               ex_body;    //error processing context object
exception OBJ_ADAPTER               ex_body;    //failure detected by object adapter
exception DATA_CONVERSION           ex_body;    //data conversion error
 exception OBJECT_NOT_EXIST         ex_body;    // non-existent object, delete reference
exception TRANSACTION_REQUIRED      ex_body;
exception TRANSACTION_ROLLEDBACK    ex_body;
exception INVALID_TRANSACTION       ex_body;
```

A.1 Object Non-Existence

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

A.2 Transaction exceptions

The TRANSACTION_REQUIRED exception indicates that the request carried a null transaction context, but an active transaction is required.The TRANSACTION_ROLLEDBACK exception indicates that the transaction associated with the request has already been rolled back or marked to roll back. Thus, the requested operation either could not be performed or was not performed because further computation on behalf of the transaction would be fruitless.

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

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Annex B

Typecode encoding in the CORBA specification

(This annex does not forms an integral part of this Recommendation | International Standard)

The ODP IDL interface for Typecodes in CORBA is:

```
module CORBA {
   enum TCKind {
      tk_null, tk_void,
      tk_short, tk_long, tk_ushort, tk_ulong,
      tk_float, tk_double, tk_boolean, tk_char,
      tk_octet, tk_any, tk_TypeCode, tk_Principal, tk_objref,
      tk_struct, tk_union, tk_enum, tk_string,
      tk_sequence, tk_array, tk_alias, tk_except,
      tk_longlong, tk_ulonglong, tk_longdouble,
      tk_wchar, tk_wstring, tk_fixed_pt
      };
   interface TypeCode {
      exception              Bounds{};
      exception              BadKind{};

      // for all TypeCode kinds
      boolean                equal (in TypeCode tc);
      TCkind                 kind ();

      // for tk_objref, tk_struct, tk_union, tkenum, tk_alias, and tk_except
      RepositoryId           id () raises (BadKind) ;

      // for tk_objref, tk_struct, tk_union, tkenum, tk_alias, and tk_except
      Identifier             name () raises (BadKind) ;

      // for tk_struct, tk_union, tk_enum, and tk_except
      unsigned long          member_count () raises (BadKind);
      Identifier             member_name (in unsigned long index)
                                  raises (BadKind, Bounds);

      // for tk_struct, tk_union, and tk_except
      TypeCode               member_type (in unsigned long index)
                                  raises (BadKinds, Bounds);

      //for tk_union
      any                    member_label (in unsigned long index)
                                  raises (BadKind, Bounds);
      TypeCode               discriminator_type () raises (BadKind);
      long                   default_index () raises (BadKind);
```


Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
        //for tk_string, tk_sequence, and tk_array
        unsigned long length () raises (BadKind);

        //for tk_sequence, tk_array, and tk_alias
        TypeCode               content_type () raises (BadKind);

        // deprecated interface
        long                   parameter_count ();
        any                    parameter (in long index) raises (Bounds) ;
    };
```

With the above operations, any Type Code can be decomposed into its constituent parts.

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

© ISO/IEC

ICS 35.100.01

Price based on 31 pages

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS

Not for Resale