INTERNATIONAL STANDARD

ISO/IEC 19516

First edition
2020-02

Information technology — Object management group — Interface definition language (IDL) 4.2

Reference number
ISO/IEC 19516:2020(E)

Contents

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.

The procedures used to develop this document and those intended for its further maintenance are described in the ISO/IEC Directives, Part 1. In particular, the different approval criteria needed for the different types of document should be noted (see www.iso.org/directives).

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. ISO and IEC shall not be held responsible for identifying any or all such patent rights. Details of any patent rights identified during the development of the document will be in the Introduction and/or on the ISO list of patent declarations received (see www.iso.org/patents) or the IEC list of patent declarations received (see http://patents.iec.ch).

Any trade name used in this document is information given for the convenience of users and does not constitute an endorsement.

For an explanation of the voluntary nature of standards, the meaning of ISO specific terms and expressions related to conformity assessment, as well as information about ISO's adherence to the World Trade Organization (WTO) principles in the Technical Barriers to Trade (TBT) see www.iso.org/iso/foreword.html.

This document was prepared by the Object Management Group (OMG) (as the OMG specification for Interface Definition Language (IDL), v4.2) and drafted in accordance with its editorial rules. It was adopted, under the JTC 1 PAS procedure, by Joint Technical Committee ISO/IEC JTC 1, *Information technology*.

This document is related to:

- ITU-T Recommendation X.902 (1995) | ISO/IEC 10746-2:1995, Information Technology — Open Distributed Processing — Reference Model: Foundations
- ITU-T Recommendation X.903 (1995) | ISO/IEC 10746-3:1995, Information Technology — Open Distributed Processing — Reference Model: Architecture
- ITU-T Recommendation X.920 (1997) | ISO/IEC 14750:1997, Information Technology — Open Distributed Processing — Interface Definition Language

Apart from this Foreword, the text of this document is identical with that for the OMG specification for Interface Definition Language (IDL), v4.2.

Any feedback or questions on this document should be directed to the user’s national standards body. A complete listing of these bodies can be found at www.iso.org/members.html.

Introduction

The rapid growth of distributed processing has led to a need for a coordinating framework for this standardization and ITU-T Recommendations X.901-904 | ISO/IEC 10746, 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.

RM-ODP Part 2 (ISO RM-ODP Part 2 (ISO/IEC 10746-2) defines the foundational concepts and modeling framework for describing distributed systems. The scopes and objectives of the RM-ODP Part 2 and the UML, while related, are not the same and, in a number of cases, the RM-ODP Part 2 and the UML specification use the same term for concepts which are related but not identical (e.g., interface). Nevertheless, a specification using the Part 2 modeling concepts can be expressed using UML with appropriate extensions (using stereotypes, tags, and constraints).

RM-ODP Part 3 (ISO/IEC 10746-3) specifies a generic architecture of open distributed systems, expressed using the foundational concepts and framework defined in Part 2. Given the relation between UML as a modeling language and Part 3 of the RM-ODP standard, it is easy to show that UML is suitable as a notation for the individual viewpoint specifications defined by the RM-ODP.

This International Standard defines a method for automating the counting of Function Points that is generally consistent with the Function Point Counting Practices Manual, Release 4.3.1 (IFPUG CPM) produced by the International Function Point Users Group (IFPUG). Guidelines in this International Standard may differ from those in the IFPUG CPM at points where subjective judgments have to be replaced by the rules needed for automation. The IFPUG CPM was selected as the anchor for this International Standard because it is the most widely used functional measurement specification with a large supporting infrastructure maintained by a professional organization.

Information technology — Object management group — Interface definition language (IDL) 4.2

1 Scope

1.1 Overview

This International Standard specifies the OMG Interface Definition Language (IDL). IDL is a descriptive language used to define data types and interfaces in a way that is independent of the programming language or operating system/processor platform.

The IDL specifies only the syntax used to define the data types and interfaces. It is normally used in connection with other standards that further define how these types/interfaces are utilized in specific contexts and platforms:

- Separate "language mapping" standards define how the IDL-defined constructs map to specific programming languages, such as, C/C++, Java, C#, etc.
- Separate "serialization" standards define how data objects and method invocations are serialized into a format suitable for network transmission.
- Separate "middleware" standards, such as, DDS or CORBA leverage the IDL to define data-types, services, and interfaces.

The description of IDL grammar uses a syntax notation that is similar to Extended Backus-Naur Format (EBNF).

2 Conformance Criteria

This International Standard defines IDL such that it can be referenced by other standards. It contains no independent conformance points. It is up to the standards that depend on this International Standard to define their own conformance criteria. However, the general organization of the clauses (by means of atomic building blocks and profiles that group them) is intended to ease conformance description and scoping. That means that no standard using IDL 4.0 will be forced to be compliant with IDL constructs that are not relevant in its usage of IDL.

Conformance to this International Standard must follow these rules:

1. Future standards that use IDL shall reference this IDL International Standard or a future revision thereof.
2. Future revisions of current standards that use IDL may reference this IDL International Standard or a future revision thereof.
3. Reference to this International Standard shall result in a selection of building blocks possibly complemented by groups of annotations.

   a. All selected building blocks shall be supported entirely.

   b. Selected annotations shall be either supported as described in 8.2.2 Rules for Using Standardized Annotations, or fully ignored. In the latter case, the IDL-dependent standard shall not define a specific annotation, either with the same name and another meaning or with the same meaning and another name.

3 Normative References

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

- ISO/IEC 14882:2003, *Information Technology — Programming languages — C++*
- [RFC2119] IETF RFC 2119, "*Key words for use in RFCs to Indicate Requirement Levels*", S. Bradner, March 1997. Available from http://ietf.org/rfc/rfc2119
- [CORBA] Common Object Request Broker Architecture. OMG specification: formal/2012-11-12 (part1), formal/2012-11-14 (part2), formal/2012-11-16 (part3)

The following referenced documents were used as input to this International Standard:

- [DDS-XTypes]. Extensible and Dynamic Topic Types for DDS, Version 1.2. Available from: http://www.omg.org/spec/DDS-XTypes/1.2
- [DDS]. Data Distribution Service, Version 1.4. Available from: http://www.omg.org/spec/DDS/1.4
- [DDS-RPC]. Remote Procedure Call over DDS, Version 1.0. Available from: http://www.omg.org/spec/DDS-RPC/1.0

4 Terms and Definitions

In this International Standard:

- A building block is a consistent set of IDL rules that together form a piece of IDL functionality. Building blocks are atomic, meaning that if selected, they must be totally supported. Building blocks are described in Clause 7, IDL Syntax and Semantics.
- A group of annotations is a consistent set of annotations, expressed in IDL. Groups of annotations are described in Clause 8, Standardized Annotations.
- A profile is a selection of building blocks possibly complemented with groups of annotations that determines a specific IDL usage. Profiles are described in Clause 9, Profiles.

5 Symbols

The following abbreviations are used throughout this International Standard.

Acronym	Meaning
ASCII	American Standard Code for Information Interchange
BIPM	Bureau International des Poids et Mesures
CCM	CORBA Component Model
CORBA	Common Object Request Broker Architecture
DDS	Data Distribution Service
EBNF	Extended Backus Naur Form

Acronym	Meaning
IDL	Interface Definition Language
ISO	International Organization for Standardization
LwCCM	Lightweight CCM
OMG	Object Management Group
ORB	Object Request Broker
XTypes	eXtensible and dynamic topic Types (for DDS)

6 Additional Information

6.1 Acknowledgments

The following companies submitted this International Standard:

— Thales

— RTI

The following companies supported this International Standard:

— Mitre

— Northrop Grumman

— Remedy IT

6.2 History

Historically, IDL was designed to specify CORBA interfaces and subsequently CORBA components. For this reason the IDL standard was embedded in the CORBA documentation. However its expressive power made it very suitable for defining non-CORBA interfaces and data types. Consequently, it was used in the DDS standard and extended to support that usage. In recognition of these new usages, and expected future ones, IDL was separated into its own stand-alone standard, independent of its use by specific middleware technologies.

This International Standard completes the definition of IDL as a separate standard, an effort started with IDL 3.5.

IDL 3.5 gathered in a single standard all the CORBA-dedicated IDL, formerly specified as a collection of clauses within the CORBA 3 standard.

IDL 4.0 extended that corpus with the other source for IDL definitions, namely "Extensible and Dynamic Topic Types for DDS" in order to group all IDL constructs in a single comprehensive standard. It organized the IDL description into modular "Building Blocks" so that different levels of compliance are easier to specify and that future evolutions, if needed, can be made without side-effects on existing IDL usages.

IDL 4.1 improved the definition of the bitset type, added a bitmap type, and resolved some inconsistencies in the grammar.

IDL 4.2 added support for 8-bit integer types, added size-explicit keywords for integer types, enhanced the readability, and reordered the building blocks to follow a logical dependency progression.

7 IDL Syntax and Semantics

7.1 Overview

This clause describes OMG Interface Definition Language (IDL) middleware[1]-agnostic semantics[2] and defines the syntax for IDL grammatical constructs.

OMG IDL is a language that allows unambiguous specification of the interfaces that client objects[3] may use and (server) object implementations provide as well as all needed related constructs such as exceptions and data types. Data types are needed to specify parameters and return value of interfaces' operations. They can be used also as first class constructs.

IDL is a purely descriptive language. This means that actual programs that use these interfaces or create the associated data types cannot be written in IDL, but in a programming language, for which mappings from IDL constructs have been defined. The mapping of IDL constructs to a programming language will depend on the facilities available in that programming language. For example, an IDL exception might be mapped to a structure in a language that has no notion of exceptions, or to an exception in a language that does. The binding of IDL constructs to several programming languages is described in separate standards.

The clause is organized as follows:

- The description of IDL's lexical conventions is presented in 7.2 Lexical Conventions.
- A description of IDL preprocessing is presented in 7.3 Preprocessing
- The grammar itself is presented in 7.4 IDL Grammar
- The scoping rules for identifiers in an IDL standard are described in 7.5.

IDL-specific pragmas may appear anywhere in a standard; the textual location of these pragmas may be semantically constrained by a particular implementation.

A source file containing specifications written in IDL shall have a "**.idl**" extension.

The description of IDL grammar uses a syntax notation that is similar to Extended Backus-Naur Format (EBNF). However, to allow composition of specific parts of the description, while avoiding redundancy; a new operator (**::+**) has been added. This operator allows adding alternatives to an existing definition. For example, assuming the rule **x ::= y**, the rule **x ::+ z** shall be interpreted as **x ::= y | z**.

Table 7.1 lists the symbols used in this EBNF format and their meaning.

[1] In this document the word *middleware* refers to any piece of software that will make use of IDL-derived artifacts. CORBA and DDS implementations are examples of middleware. The word *compiler* refers to any piece of software that produces these IDL-derived artifacts based on an IDL specification.

[2] I.e., abstract semantics that is applicable to all IDL usages. When needed, middleware-specific interpretations of that abstract semantics will be given afterwards in dedicated clauses.

[3] Accordingly, *client objects* should be understood here as abstract clients, i.e., entities invoking operations provided by object implementations, regardless of the means used to perform this invocation or even whether those implementations are co-located or remotely accessible.

Table 7.1 — IDL EBNF

Symbol	Meaning
`::=`	Is defined to be *(left part of the rule is defined to be right part of the rule)*
`|`	Alternatively
`::+`	Is added as alternative *(left part of the rule is completed with right part of the rule as a new alternative)*
`<text>`	Nonterminal
`"text"`	Literal
`*`	The preceding syntactic unit can be repeated zero or more times
`+`	The preceding syntactic unit must be repeated at least once
`{}`	The enclosed syntactic units are grouped as a single syntactic unit
`[]`	The enclosed syntactic unit is optional – may occur zero or one time

7.2 Lexical Conventions

This sub clause[4] presents the lexical conventions of IDL. It defines tokens in an IDL specification and describes comments, identifiers, keywords, and literals - integer, character, and floating point constants and string literals.

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

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

IDL uses the ASCII character set, except for string literals and character literals, which use the ISO Latin-1 (8859-1) character set. The ISO Latin-1 character set is divided into alphabetic characters (letters) digits, graphic characters, the space (blank) character, and formatting characters. Table 7.2 shows the ISO Latin-1 alphabetic characters; upper and lower case equivalences are paired. The ASCII alphabetic characters are shown in the left-hand column of Table 7.2.

Table 7.2 — Characters

Char.	Description	Char.	Description
Aa	Upper/Lower-case A	Àà	Upper/Lower-case A with grave accent
Bb	Upper/Lower-case B	Áá	Upper/Lower-case A with acute accent
Cc	Upper/Lower-case C	Ââ	Upper/Lower-case A with circumflex accent
Dd	Upper/Lower-case D	Ãã	Upper/Lower-case A with tilde
Ee	Upper/Lower-case E	Ää	Upper/Lower-case A with dieresis
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

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

Char.	Description	Char.	Description
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 dieresis
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 dieresis
Qq	Upper/Lower-case Q	Ññ	Upper/Lower-case N with tilde
Rr	Upper/Lower-case R	Òò	Upper/Lower-case O with grave accent
Ss	Upper/Lower-case S	Óó	Upper/Lower-case O with acute accent
Tt	Upper/Lower-case T	Ôô	Upper/Lower-case O with circumflex accent
Uu	Upper/Lower-case U	Õõ	Upper/Lower-case O with tilde
Vv	Upper/Lower-case V	Öö	Upper/Lower-case O with dieresis
Ww	Upper/Lower-case W	Øø	Upper/Lower-case O with oblique stroke
Xx	Upper/Lower-case X	Ùù	Upper/Lower-case U with grave accent
Yy	Upper/Lower-case Y	Úú	Upper/Lower-case U with acute accent
Zz	Upper/Lower-case Z	Ûû	Upper/Lower-case U with circumflex accent
		Üü	Upper/Lower-case U with dieresis
		ß	Lower-case German sharp S
		ÿ	Lower-case Y with dieresis

Table 7.3 lists the decimal digit characters.

Table 7.3 — Decimal Digits

0	1	2	3	4	5	6	7	8	9

Table 7.4 shows the graphic characters.

Table 7.4 — 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

Char.	Description	Char.	Description
'	apostrophe	§	section/paragraph sign
(	left parenthesis	¨	dieresis
)	right parenthesis	©	copyright sign
*	asterisk	ª	feminine ordinal indicator
+	plus sign	«	left angle quotation mark
,	comma	¬	not sign
-	hyphen, minus sign		soft hyphen
.	period, full stop	®	registered trade mark sign
/	solidus	¯	macron
:	colon	°	ring above, degree sign
;	semicolon	±	plus-minus sign
<	less-than sign	²	superscript two
=	equals sign	³	superscript three
>	greater-than sign	´	acute
?	question mark	µ	micro
@	commercial at	¶	pilcrow
[	left square bracket	·	middle dot
\	reverse solidus	¸	cedilla
]	right square bracket	¹	superscript one
^	circumflex	º	masculine ordinal indicator
_	low line, underscore	»	right angle quotation mark
`	grave	¼	vulgar fraction 1/4
{	left curly bracket	½	vulgar fraction 1/2
|	vertical line	¾	vulgar fraction 3/4
}	right curly bracket	¿	inverted question mark
~	tilde	×	multiplication sign
		÷	division sign

The formatting characters are shown in Table 7.5.

Table 7.5 — Formatting Characters

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

7.2.1 Tokens

There are five kinds of tokens: identifiers, keywords, literals, operators, and other separators.

Blanks, horizontal and vertical tabs, newlines, form feeds, 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.

7.2.2 Comments

The characters /* start a comment, which terminates with the characters */. These comments do not nest.

The characters // start a comment, which terminates at the end of the line on which they occur.

The comment characters //, /*, and */ have no special meaning within a // comment and are treated just like other characters. Similarly, the comment characters // and /* have no special meaning within a /* comment.

Comments may contain alphabetic, digit, graphic, space, horizontal tab, vertical tab, form feed, and newline characters.

7.2.3 Identifiers

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

IDL identifiers are case insensitive. However, all references to a definition must use the same case as the defining occurrence. This allows natural mappings to case-sensitive languages.

7.2.3.1 Collision Rules

When comparing two identifiers to see if they collide:

- Upper- and lower-case letters are treated as the same letter. Table 7.2 defines the equivalence mapping of upper- and lower-case letters.
- All characters are significant.

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

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

EXAMPLE

```
module M {
      typedef long Foo;
      const long thing = 1;
      interface thing {                  // Error: reuse of identifier thing
           void doit (
                    in Foo foo           // Error: Foo and foo collide…
                    );                   //          … and refer to different things
           readonly attribute long Attribute; // Error: Attribute collides with keyword…
                                         //           … attribute
           };
      };
```

7.2.3.2 Escaped Identifiers

As all languages, IDL uses some reserved words called *keywords* (see 7.2.4).

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

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

EXAMPLE

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

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

7.2.4 Keywords

The identifiers listed in Table 7.6 are reserved for use as keywords and may not be used for another purpose, unless escaped with a leading underscore.

Table 7.6 — All IDL Keywords

abstract	any	alias	attribute	bitfield
bitmask	bitset	boolean	case	char
component	connector	const	consumes	context
custom	default	double	exception	emits
enum	eventtype	factory	FALSE	finder
fixed	float	getraises	home	import
in	inout	interface	local	long
manages	map	mirrorport	module	multiple
native	Object	octet	oneway	out
primarykey	private	port	porttype	provides
public	publishes	raises	readonly	setraises
sequence	short	string	struct	supports
switch	TRUE	truncatable	typedef	typeid
typename	typeprefix	unsigned	union	uses
ValueBase	valuetype	void	wchar	wstring
int8	uint8	int16	int32	int64
uint16	uint32	uint64		

Keywords must be written exactly as shown in the above list. Identifiers that collide with keywords (see 7.2.3) are illegal. For example, **boolean** is a valid keyword; **Boolean** and **BOOLEAN** are illegal identifiers.

EXAMPLE

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

NOTE As the IDL grammar is now organized in building blocks that dedicated profiles may include or not, some of these keywords may be irrelevant to some profiles. Each building block description (see 7.4) includes the set of keywords that are specific to it. The minimum set of keywords for a given profile results from the union of all the ones of the included building blocks. However, to avoid unnecessary confusion for readers of IDL and to allow IDL compilers supporting several profiles in a single tool, it is recommended that IDL specifications avoid using any of the keywords listed in Table 7.6.

7.2.5 Other Characters Recognized by IDL

IDL specifications use the characters shown in Table 7.7 as punctuation.

Table 7.7 — Punctuation

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

In addition, the tokens listed in Table 7.8 are used by the preprocessor.

Table 7.8 — Tokens

#	##	!	||	&&

7.2.6 Literals

This sub clause describes the following literals:

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

7.2.6.1 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 and thus are not allowed in an octal integer literal.

A sequence of digits preceded by **0x** (or **0X**) is taken to be a *hexadecimal integer* (base sixteen). The hexadecimal digits include **a** (or **A**) through **f** (or **F**) with decimal values ten through fifteen, respectively.

For example, the number twelve can be written **12**, **014**, or **0XC**.

7.2.6.2 Character Literals

7.2.6.2.1 Wide and Non-wide Characters

Character literals may have type **char** (non-wide character) or **wchar** (wide character).

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

A **char** is an 8-bit quantity with a numerical value between 0 and 255 (decimal). The value of a space, alphabetic, digit, or graphic character literal is the numerical value of the character as defined in the ISO Latin-1 (8859-1) character set standard (see Table 7.2, Table 7.3, and Table 7.4). The value of a **null** is 0. The value of a formatting character literal is the numerical value of the character as defined in the ISO 646 standard (see Table 7.5). The meaning of all other characters is implementation-dependent.

A **wchar** (wide character) is intended to encode wide characters from any character set. Its size is implementation dependent.

A character literal is one or more characters enclosed in single quotes, as in:

```
const char C1 = 'X';
```

Wide character literals have in addition an **L** prefix, as in:

```
const wchar C2 = L'X';
```

Attempts to assign a wide character literal to a non-wide character constant, or to assign a non-wide character literal to a wide character constant, shall be treated as an error.

7.2.6.2.2 Escape Sequences to Represent Character Literals

Nongraphic characters shall be represented using escape sequences as defined below in Table 7.9. Note that escape sequences shall be used to represent single quote and backslash characters in character literals.

Table 7.9 — Escape Sequences

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

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

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

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

The escape **\uhhhh** consists of a backslash followed by the character **u**, followed by one, two, three, or four hexadecimal digits. This represents a Unicode character literal. For example, the literal **\u002E** represents the Unicode period '.' character and the literal **\u3BC** represents the Unicode Greek small letter 'µ' (mu). The **\u** escape is valid only with **wchar** and **wstring** types. Because a wide string literal is defined as a sequence of wide character literals, a sequence of **\u** literals can be used to define a wide string literal.

Attempts to set a char type to a **\u** defined literal or a string type to a sequence of **\u** literals result in an error.

7.2.6.3 String Literals

Strings are null-terminated sequences of characters. Strings are of type **string** if they are made of non-wide characters or **wstring** (wide string) if they are made of wide characters.

A string literal is a sequence of character literals with the exception of the character with numeric value 0, surrounded by double quotes, as in:

```
const string S1 = "Hello";
```

Wide string literals have in addition an **L** prefix, for example:

```
const wstring S2 = L"Hello";
```

Both wide and non-wide string literals must be specified using characters from the ISO Latin-1 (8859-1) character set. A string literal shall not contain the character '\0'. A wide string literal shall not contain the wide character with value zero.

Adjacent string literals are concatenated. Characters in concatenated strings are kept distinct. For example, **"\xA" "B"** contains the two characters '**\xA**' and '**B**' after concatenation (and not the single hexadecimal character **'\xAB'**).

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

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

7.2.6.4 Floating-point Literals

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

7.2.6.5 Fixed-Point Literals

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

7.3 Preprocessing

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

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

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

A preprocessing token is an IDL token (see 7.2.1), 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 IDL specifications. Text in files included with a **#include** directive is treated as if it appeared in the including file.

NOTE Generating code for included files is an IDL compiler implementation-specific issue. To support separate compilation, IDL compilers may not generate code for included files, or do so only if explicitly instructed.

7.4 IDL Grammar

The grammar for a well-formed IDL specification is described by rules expressed in Extended Backus Naur Form (EBNF) completed to support rule extensions as explained in 7.1. Table 7.1 gathers all the symbols used in rules.

These rules are grouped in atomic *building blocks* that will be themselves grouped to form dedicated *profiles*. Atomic means that they cannot be split (in other words a given profile will contain or not a given building block, but never just parts of it).

In all the building block descriptions, the normative rules are grouped in a sub clause entitled "Syntax" and written in **Arial bold**. They are then detailed in a sub clause entitled "Explanations and Semantics", where they are copied to ease understanding. These copies are actually hyperlinks to the originals and are written in ***Arial bold-italics***.

In all the rules, the following non-terminals pre-exist and are not detailed.

Table 7.10 — List of pre-existing non-terminals in IDL rules

Token	Explanation
<identifier>	A valid identifier, as explained in 7.2.3 Identifiers
<integer_literal>	A valid integer literal as explained in7.2.6.1, Integer Literals
<string_literal>	A valid string literal as explained in 7.2.6.3 String Literals
<wide_string_literal>	A valid wide string literal as explained in 7.2.6.3, String Literals
<character_literal>	A valid character literal as explained in 7.2.6.2, Character Literals
<wide_character_literal>	A valid wide character literal as explained in7.2.6.2, Character Literals
<fixed_pt_literal>	A valid fixed point literal as explained in 7.2.6.5 Fixed-Point Literals
<floating_pt_literal>	A valid floating point literal as explained in 7.2.6.4 Floating Point Literals

7.4.1 Building Block Core Data Types

7.4.1.1 Purpose

This building block constitutes the core of any IDL specialization (all other building blocks assume that this one is included). It contains the syntax rules that allow defining most data types and the syntax rules that allow IDL basic structuring (i.e., modules). Since it is the only mandatory building block, it also contains the root nonterminal <**specification**> for the grammar itself.

7.4.1.2 Dependencies with other Building Blocks

This building block is the root for all other building blocks and requires no other ones.

7.4.1.3 Syntax

The following set of rules form the building block:

```
(1)  <specification>          ::=   <definition>
(2)  <definition>             ::=   <module_dcl> ";"
                              |     <const_dcl> ";"
                              |     <type_dcl> ";"
(3)  <module_dcl>             ::=   "module" <identifier> "{" <definition>+ "}"
(4)  <scoped_name>            ::=   <identifier>
                              |     "::" <identifier>
                              |     <scoped_name> "::" <identifier>
(5)  <const_dcl>              ::=   "const" <const_type> <identifier> "=" <const_expr>
(6)  <const_type>             ::=   <integer_type>
                              |     <floating_pt_type>
                              |     <fixed_pt_const_type>
                              |     <char_type>
```

```
                                   |     <wide_char_type>
                                   |     <boolean_type>
                                   |     <octet_type>
                                   |     <string_type>
                                   |     <wide_string_type>
                                   |     <scoped_name>
(7)  <const_expr>                  ::=   <or_expr>
(8)  <or_expr>                     ::=   <xor_expr>
                                   |     <or_expr> "|" <xor_expr>
(9)  <xor_expr>                    ::=   <and_expr>
                                   |     <xor_expr> "^" <and_expr>
(10)  <and_expr>                   ::=   <shift_expr>
                                   |     <and_expr> "&" <shift_expr>
(11) <shift_expr>                  ::=   <add_expr>
                                   |     <shift_expr> ">>" <add_expr>
                                   |     <shift_expr> "<<" <add_expr>
(12) <add_expr>                    ::=   <mult_expr>
                                   |     <add_expr> "+" <mult_expr>
                                   |     <add_expr> "-" <mult_expr>
(13) <mult_expr>                   ::=   <unary_expr>
                                   |     <mult_expr> "*" <unary_expr>
                                   |     <mult_expr> "/" <unary_expr>
                                   |     <mult_expr> "%" <unary_expr>
(14) <unary_expr>                  ::=   <unary_operator> <primary_expr>
                                   |     <primary_expr>
(15) <unary_operator>              ::=   "-"
                                   |     "+"
                                   |     "~"
(16) <primary_expr>                ::=   <scoped_name>
                                   |     <literal>
                                   |     "(" <const_expr> ")"
(17) <literal>                     ::=   <integer_literal>
                                   |     <floating_pt_literal>
                                   |     <fixed_pt_literal>
                                   |     <character_literal>
                                   |     <wide_character_literal>
                                   |     <boolean_literal>
                                   |     <string_literal>
                                   |     <wide_string_literal>
(18) <boolean_literal>             ::=   "TRUE"
                                   |     "FALSE"
(19) <positive_int_const>          ::=   <const_expr>
(20) <type_dcl>                    ::=   <constr_type_dcl>
                                   |     <native_dcl>
                                   |     <typedef_dcl>
(21) <type_spec>                   ::=   <simple_type_spec>
(22) <simple_type_spec>            ::=   <base_type_spec>
                                   |     <scoped_name>
(23) <base_type_spec>              ::=   <integer_type>
                                   |     <floating_pt_type>
                                   |     <char_type>
                                   |     <wide_char_type>
                                   |     <boolean_type>
                                   |     <octet_type>
(24) <floating_pt_type>            ::=   "float"
                                   |     "double"
                                   |     "long" "double"
(25) <integer_type>                ::=   <signed_int>
                                   |     <unsigned_int>
(26) <signed_int>                  ::=   <signed_short_int>
                                   |     <signed_long_int>
                                   |     <signed_longlong_int>
(27) <signed_short_int>            ::=   "short"
(28) <signed_long_int>             ::=   "long"
(29) <signed_longlong_int>         ::=   "long" "long"
(30) <unsigned_int>                ::=   <unsigned_short_int>
                                   |     <unsigned_long_int>
```

```
                                      |      <unsigned_longlong_int>
(31)  <unsigned_short_int>           ::=     "unsigned" "short"
(32)  <unsigned_long_int>            ::=     "unsigned" "long"
(33)  <unsigned_longlong_int>        ::=     "unsigned" "long" "long"
(34)  <char_type>                    ::=     "char"
(35)  <wide_char_type>               ::=     "wchar"
(36)  <boolean_type>                 ::=     "boolean"
(37)  <octet_type>                   ::=     "octet"
(38)  <template_type_spec>           ::=     <sequence_type>
                                      |      <string_type>
                                      |      <wide_string_type>
                                      |      <fixed_pt_type>
(39)  <sequence_type>                ::=     "sequence" "<" <type_spec> "," <positive_int_const> ">"
                                      |      "sequence" "<" <type_spec> ">"
(40)  <string_type>                  ::=     "string" "<" <positive_int_const> ">"
                                      |      "string"
(41)  <wide_string_type>             ::=     "wstring" "<" <positive_int_const> ">"
                                      |       "wstring"
(42)  <fixed_pt_type>                ::=     "fixed" "<" <positive_int_const> "," <positive_int_const> ">"
(43)  <fixed_pt_const_type>          ::=     "fixed"
(44)  <constr_type_dcl>              ::=     <struct_dcl>
                                      |      <union_dcl>
                                      |      <enum_dcl>
(45)  <struct_dcl>                   ::=     <struct_def>
                                      |      <struct_forward_dcl>
(46)  <struct_def>                   ::=     "struct" <identifier> "{" <member>+ "}"
(47)  <member>                       ::=     <type_spec> <declarators> ";"
(48)  <struct_forward_dcl>           ::=     "struct" <identifier>
(49)  <union_dcl>                    ::=     <union_def>
                                      |      <union_forward_dcl>
(50)  <union_def>                    ::=     "union" <identifier> "switch" "(" <switch_type_spec> ")"
                                             "{" <switch_body> "}"
(51)  <switch_type_spec>             ::=     <integer_type>
                                      |      <char_type>
                                      |      <boolean_type>
                                      |      <scoped_name>
(52)  <switch_body>                  ::=     <case>+
(53)  <case>                         ::=     <case_label>+ <element_spec> ";"
(54)  <case_label>                   ::=     "case" <const_expr> ":"
                                      |      "default" ":"
(55)  <element_spec>                 ::=     <type_spec> <declarator>
(56)  <union_forward_dcl>            ::=     "union" <identifier>
(57)  <enum_dcl>                     ::=     "enum" <identifier>
                                             "{" <enumerator> { "," <enumerator> } * "}"
(58)  <enumerator>                   ::=     <identifier>
(59)  <array_declarator>             ::=     <identifier> <fixed_array_size>+
(60)  <fixed_array_size>             ::=     "[" <positive_int_const> "]"
(61)  <native_dcl>                   ::=     "native" <simple_declarator>
(62)  <simple_declarator>            ::=     <identifier>
(63)  <typedef_dcl>                  ::=     "typedef" <type_declarator>
(64)  <type_declarator>              ::=     { <simple_type_spec>
                                             | <template_type_spec>
                                             | <constr_type_dcl>
                                             } <any_declarators>
(65)  <any_declarators>             ::=     <any_declarator> { "," <any_declarator> }*
(66)  <any_declarator>              ::=     <simple_declarator>
                                      |      <array_declarator>
(67)  <declarators>                  ::=     <declarator> { "," <declarator> }*
(68)  <declarator>                   ::=     <simple_declarator>
```

7.4.1.4 Explanations and Semantics

7.4.1.4.1 IDL Specification

An IDL specification consists of one or more definitions.

```
(1) <specification>      ::=      <definition>+
```

In this building block, supported definitions are: module definitions, constant definitions and (data) type definitions as expressed in the following rule:

```
(2)    <definition>              ::= <module_dcl> ";"
                                 |   <const_dcl> ";"
                                 |   <type_dcl> ";"
```

7.4.1.4.2 Modules

A module is a grouping construct. Its definition satisfies the following rule:

```
(3)    <module_dcl>              ::= "module" <identifier> "{" <definition>+ "}"
```

A module is declared with:

- The **module** keyword.
- An identifier (<**identifier**>) which is the name of the module. That name is then used to scope embedded IDL identifiers.
- A list of at least one definition (<**definition**>+) enclosed within braces ({}). Those definitions form the module body.

A scoped name is built according to the following rule:

```
(4)    <scoped_name>             ::= <identifier>
                                 |   "::" <identifier>
                                 |   <scoped_name> "::" <identifier>
```

For more details on scoping rules, see 7.5.

An IDL module can be reopened, which means that when a module declaration is encountered with a name already given to an existing module, all the enclosed definitions are appended to that existing module: the two module statements are thus considered as subsequent parts of the same module description.

7.4.1.4.3 Constants

Well-formed constants shall follow the following rules:

```
(5)    <const_dcl>               ::= "const" <const_type> <identifier> "=" <const_expr>
(6)    <const_type>              ::= <integer_type>
                                 |   <floating_pt_type>
                                 |   <fixed_pt_const_type>
                                 |   <char_type>
                                 |   <wide_char_type>
                                 |   <boolean_type>
                                 |   <octet_type>
                                 |   <string_type>
                                 |   <wide_string_type>
                                 |   <scoped_name>
(7)    <const_expr>              ::= <or_expr>
(8)    <or_expr>                 ::= <xor_expr>
                                 |   <or_expr> "|" <xor_expr>
(9)    <xor_expr>                ::= <and_expr>
                                 |   <xor_expr> "^" <and_expr>
(10)   <and_expr>                ::= <shift_expr>
                                 |   <and_expr> "&" <shift_expr>
(11)   <shift_expr>              ::= <add_expr>
                                 |   <shift_expr> ">>" <add_expr>
                                 |   <shift_expr> "<<" <add_expr>
```

```
(12)   <add_expr>           ::= <mult_expr>
                            |   <add_expr> "+" <mult_expr>
                            |   <add_expr> "-" <mult_expr>

(13)   <mult_expr>          ::= <unary_expr>
                            |   <mult_expr> "*" <unary_expr>
                            |   <mult_expr> "/" <unary_expr>
                            |   <mult_expr> "%" <unary_expr>

(14)   <unary_expr>         ::= <unary_operator> <primary_expr>
                            |   <primary_expr>

(15)   <unary_operator>     ::= "-"
                            |   "+"
                            |   "~"

(16)   <primary_expr>       ::= <scoped_name>
                            |   <literal>
                            |   "(" <const_expr> ")"

(17)   <literal>            ::= <integer_literal>
                            |   <floating_pt_literal>
                            |   <fixed_pt_literal>
                            |   <character_literal>
                            |   <wide_character_literal>
                            |   <boolean_literal>
                            |   <string_literal>
                            |   <wide_string_literal>

(18)   <boolean_literal>    ::= "TRUE"
                            |   "FALSE"

(19)   <positive_int_const> ::= <const_expr>
```

According to those rules, a constant is defined by:

- The **const** keyword.
- A type declaration, which shall denote a type suitable for a constant (**<const_type>**), i.e.,:
  - Either one of the following: **<integer_type>**, **<floating_pt_type>, <fixed_pt_const_type>, <char_type>**, **<wide_char_type>**, **<boolean_type>**, **<octet_type>**, **<string_type>**, **<wide_string_type>**, or a previously defined enumeration. For a definition of those types, see 7.4.1.4.4, Data Types.
  - Or a **<scoped_name>**, which shall be a previously defined name of one of the above.
- The name given to the constant (<**identifier**>).
- The operator =.
- A value expression (<**const_expr**>), which shall be consistent with the type declared for the constant.

For evaluating the value expression (right hand side of the constant declaration), the following rules shall be applied:

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

- If the type of a floating-point constant is **long double**, then each sub-expression of the associated constant expression is treated as a **long double**. It is an error if any sub-expression value exceeds the precision of **long double**.
- An infix operator may combine two integer types, floating point types or fixed point types, but not mixtures of these. Infix operators shall be applicable only to integer, floating point, and fixed point types.
- Integer expressions shall be evaluated based on the type of each argument of a binary operator in turn. If either argument is **unsigned long long**, it shall use **unsigned long long**. If either argument is **long long**, it shall use **long long**. If either argument is **unsigned long**, it shall use **unsigned long**. Otherwise it shall use **long**. The final result of an integer arithmetic expression shall fit in the range of the declared type of the constant; otherwise it shall be treated as an error. In addition to the integer types, the final result of an integer arithmetic expression may be assigned to an octet constant, subject to it fitting in the range for **octet** type.
- Floating point expressions shall be evaluated based on the type of each argument of a binary operator in turn. If either argument is **long double**, it shall use **long double**. Otherwise it shall use **double**. The final result of a floating point arithmetic expression shall fit in the range of the declared type of the constant; otherwise it shall be treated as an error.
- Fixed-point decimal constant expressions shall be evaluated as follows. A fixed-point literal has the apparent number of total and fractional digits. For example, **0123.450d** is considered to be **fixed<7,3>** and **3000.00d** is **fixed<6,2>.** Prefix operators do not affect the precision; a prefix + is optional, and does not change the result. The upper bounds on the number of digits and scale of the result of an infix expression, **fixed<d1 ,s1> op fixed<d2,s2>**, are shown in the following table.

Table 7.11 — Operations on fixed-point decimal constants

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

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

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

- Leading and trailing zeros shall not be considered significant. The omitted digits shall be discarded; rounding shall not be 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 shall be applicable in floating-point and fixed-point expressions.
  - The **+** unary operator shall have no effect; the **–** unary operator indicates that the sign of the following expression is inverted.
  - The ***** binary operator indicates that the two operands shall be multiplied; the **/** binary operator indicates that the first operand shall be divided by the second one; the **+** binary operator indicates that the two operands shall be added; the **–** binary operator indicates that the second operand shall be subtracted from the first one.
- Unary (**+ - ~**) and binary (*** / % + - << >> & | ^**) operators are applicable in integer expressions.

- The **+** unary operator shall have no effect; the **–** unary operator indicates that the sign of the following expression is inverted.
- The ***** binary operator indicates that the two operands shall be multiplied; the **/** binary operator indicates that the first operand shall be divided by the second one; the **+** binary operator indicates that the two operands shall be added; the **–** binary operator indicates that the second operand shall be subtracted from the first one.
- The **~** unary operator indicates that the bit-complement of the expression to which it is applied shall be generated. For the purposes of such expressions, the values are 2's complement numbers. As such, the complement can be generated as follows:

Table 7.12 — 2's complement numbers

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

- The **%** binary operator yields the remainder from the division of the first expression by the second. If the second operand of **%** is 0, the result is undefined; otherwise **(a/b)*b + a%b** is equal to **a**.
  If both operands are non-negative, then the remainder is non-negative; if not, the sign of the remainder is implementation dependent.
- The **<<** binary operator indicates that the value of the left operand shall be shifted left the number of bits specified by the right operand, with 0 fill for the vacated bits. The right operand shall be in the range 0 <= right operand < 64.
- The **>>** binary operator indicates that the value of the left operand shall be shifted right the number of bits specified by the right operand, with 0 fill for the vacated bits. The right operand shall be in the range 0 <= right operand < 64.
- The **&** binary operator indicates that the logical, bitwise **AND** of the left and right operands shall be generated.
- The **|** binary operator indicates that the logical, bitwise **OR** of the left and right operands shall be generated.
- The **^** binary operator indicates that the logical, bitwise **EXCLUSIVE-OR** of the left and right operands shall be generated.

- <**positive_int_const**> shall evaluate to a positive integer constant.

The consistency rules between the value (right hand side of the constant declaration) and the constant type declaration (left hand side) are as follows:

- Integer literals have positive integer values. Constant integer literals shall be considered **unsigned long** unless the value is too large, then they shall be considered **unsigned long long**. Unary minus shall be considered an operator, not a part of an integer literal. Only integer values can be assigned to integer type (**short, long, long long**) constants, and **octet** constants. Only positive integer values can be assigned to unsigned integer type constants. If the value of an integer constant declaration is too large to fit in the actual type of the constant on the left hand side (for example **const short s = 655592**;) or is inappropriate for the actual type of the constant (for example **const octet o = -54**;) it shall be treated as an error.
- Octet literals have integer value in the range 0…255. If the right hand side of an octet constant declaration is outside this range, it shall be treated as an error.

- An octet constant can be defined using an integer literal or an integer constant expression but values outside the range 0…255 shall be treated as an error.
- Floating point literals have floating point values. Only floating point values can be assigned to floating point type (**float, double, long double**) constants. Constant floating point literals are considered double unless the value is too large, then they are considered **long double**. If the value of the right hand side is too large to fit in the actual type of the constant to which it is being assigned, it shall be treated as an error. Truncation on the right for floating point types is OK.
- Fixed point literals have fixed point values. Only fixed point values can be assigned to fixed point type constants. If the fixed point value in the expression on the right hand side is too large to fit in the actual fixed point type of the constant on the left hand side, then it shall be treated as an error. Truncation on the right for fixed point types is OK.
- An **enum** constant can only be defined using a scoped name for the enumerator. The scoped name is resolved using the normal scope resolution rules (see 7.5). For example:

```
enum Color { red, green, blue };
const Color FAVORITE_COLOR = red;
module M {
        enum Size { small, medium, large };
        };
const M::Size MYSIZE = M::medium;
```

- The constant name for the value of an enumerated constant definition shall denote one of the enumerators defined for the enumerated type of the constant. For example:

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

7.4.1.4.4 Data Types

A data type may be either a simple type or a constructed one. Those different kinds are detailed in the following clauses.

7.4.1.4.4.1 Referencing Types

Type declarations may reference other types. These type references can be split in several categories:

- References to basic types representing primitive builtin types such as numbers and characters. These use the keyword that identifies the type.
- References to types explicitly constructed or explictly named types. These use the scoped name of the type.
- References to anonymous template types that must be instantiated with a length (e.g., strings) or a length and an element type (e.g., sequences).

These categories are detailed in the subsequent clauses.

NOTE Within this building block, anonymous types, that is, the type resulting from an instantiation of a template type (see Building Block Anonymous Types) cannot be used directly. Instead, prior to any use, template types must be given a name through a **typedef** declaration. Therefore, as expressed in the following rules, referring to a simple type can be done either using directly its name, if it is a basic type, or using a scoped name, in all other cases:

```
(21)  <type_spec>          ::= <simple_type_spec>
(22)  <simple_type_spec>   ::= <base_type_spec>
                            |  <scoped_name>
```

7.4.1.4.4.2 Basic Types

Basic types are pre-existing types that represent numbers or characters. The set of basic types is defined by the following rules:

```
(23)   <base_type_spec>          ::= <integer_type>
                                   |  <floating_pt_type>
                                   |  <char_type>
                                   |  <wide_char_type>
                                   |  <boolean_type>
                                   |  <octet_type>
(24)   <floating_pt_type>        ::= "float"
                                   |  "double"
                                   |  "long" "double"
(25)   <integer_type>            ::= <signed_int>
                                   |  <unsigned_int>
(26)   <signed_int>              ::= <signed_short_int>
                                   |  <signed_long_int>
                                   |  <signed_longlong_int>
(27)   <signed_short_int>        ::= "short"
(28)   <signed_long_int>         ::= "long"
(29)   <signed_longlong_int>     ::= "long" "long"
(30)   <unsigned_int>            ::= <unsigned_short_int>
                                   |  <unsigned_long_int>
                                   |  <unsigned_longlong_int>
(31)   <unsigned_short_int>      ::= "unsigned" "short"
(32)   <unsigned_long_int>       ::= "unsigned" "long"
(33)   <unsigned_longlong_int>   ::= "unsigned" "long" "long"
(34)   <char_type>               ::= "char"
(35)   <wide_char_type>          ::= "wchar"
(36)   <boolean_type>            ::= "boolean"
(37)   <octet_type>              ::= "octet"
```

Integer Types

IDL integer types are **short, unsigned short, long, unsigned long, long long**, and **unsigned long long** representing integer values in the range indicated below in Table 7.13.

Table 7.13 — Integer types

Integer type	Value range
N/A. See 7.4.1.3 Building Block Extended Data-Types	$-2^{7} \ldots 2^{7} - 1$
short	$-2^{15} \ldots 2^{15} - 1$
long	$-2^{31} \ldots 2^{31} - 1$
long long	$-2^{63} \ldots 2^{63} - 1$
N/A. See 7.4.1.3 Building Block Extended Data-Types	$0 \ldots 2^{8} - 1$
unsigned short	$0 \ldots 2^{16} - 1$
unsigned long	$0 \ldots 2^{32} - 1$
unsigned long long	$0 \ldots 2^{64} - 1$

Floating-Point Types

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

Char Type

IDL defines a **char** data type that is an 8-bit quantity that (1) encodes a single-byte character from any byte-oriented code set, or (2) when used in an array, encodes a multi-byte character from a multi-byte code set.

Wide Char Type

IDL defines a **wchar** data type that encodes wide characters from any character set. The size of **wchar** is implementation-dependent.

Boolean Type

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

Octet Type

The **octet** type is an opaque 8-bit quantity that is guaranteed not to undergo any change by the middleware.

7.4.1.4.4.3 Template Types

Template types are generic types that are parameterized by type of underlying elements and/or the number of elements. To be used as an actual type, such a generic definition must be *instantiated*, i.e., given parameter values, whose nature depends on the template type.

As specified in the following rule, template types are sequences (<**sequence_type**>), strings (<**string_type**>, wide strings (<**wide_string_type**>) and fixed-point numbers (<**fixed_pt_type**>).

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

Sequences

Sequences are defined according to the following syntax.

```
(39)  <sequence_type>         ::= "sequence" "<" <type_spec> "," <positive_int_const> ">"
                                |   "sequence" "<" <type_spec> ">"
```

As a template type, **sequence** has two parameters:

- The first non-optional parameter (<**type_spec**>) gives the type of each item in the sequence.
- The second optional parameter (<**positive_int_const**> is a positive integer constant that indicates the maximum size of the sequence. If it is given, the sequence is termed a *bounded* sequence. Otherwise the sequence is said *unbounded* and no maximum size is specified.

Before using a bounded or unbounded sequence, the length of the sequence must be set in a language-mapping dependent manner. If the bounded form is used, the length must be less than or equal to the maximum. Similarly after receiving a sequence, this value may be obtained in a language-mapping dependent manner.

Strings

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

```
(40)   <string_type>          ::= "string" "<" <positive_int_const> ">"
                              |   "string"
```

The argument to the string declaration is the maximum size of the string (<**positive_int_const**>). 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. The actual length of a string is set at run-time and, if the bounded form is used, must be less than or equal to the maximum.

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.

Wstrings

The **wstring** data type represents a sequence of **wchar**, except the wide character null. The type **wstring** is similar to that of type **string**, except that its element type is **wchar** instead of **char**. The syntax for defining a **wstring** is:

```
(41)   <wide_string_type>     ::= "wstring" "<" <positive_int_const> ">"
                              |   "wstring"
```

Fixed Type

The **fixed** data type represents a fixed-point decimal number of up to 31 significant digits. The syntax to declare a fixed data type is:

```
(42)   <fixed_pt_type>        ::= "fixed" "<" <positive_int_const> "," <positive_int_const> ">"
```

The first parameter is the number of total digits (up to 31), the second one the number of fractional digits, which must be less or equal to the former.

In case the fixed type specification is used in a constant declaration, those two parameters are omitted as they are automatically deduced from the constant value. The syntax is thus as follows:

```
(43)   <fixed_pt_const_type>  ::= "fixed"
```

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

7.4.1.4.4.4 Constructed Types

Constructed types are types that are created by an IDL specification.

As expressed in the following rule, structures (<**struct_dcl**>), unions (<**union_dcl**>) and enumerations (<**enum_dcl**>) are the constructed types supported in this building block:

```
(44)   <constr_type_dcl>      ::= <struct_dcl>
                              |   <union_dcl>
                              |   <enum_dcl>
```

All those constructs are presented in the following clauses.

Structures

A structure is a grouping of at least one member. The syntax to declare a structure is as follows:

```
(45)   <struct_dcl>           ::= <struct_def>
                              |   <struct_forward_dcl>
(46)   <struct_def>           ::= "struct" <identifier> "{" <member>+ "}"
(47)   <member>               ::= <type_spec> <declarators> ";"
```

```
(67)   <declarators>          ::= <declarator> { "," <declarator> }*
(68)   <declarator>           ::= <simple_declarator>
```

A structure definition comprises:

- The **struct** keyword.
- The name given to the structure (<**identifier**>).
- The list of all structure members (<**member**>+) enclosed within braces ({}). Each member (<**member**>) is defined with a type specification (<**type_spec**>) followed by a list of at least one declarator (<**declarators**>). At least one member is required.

The name of a structure defines a new legal type that may be used anywhere such a type is legal in the grammar.

NOTE Members may be of any data type, including sequences or arrays. However, except when anonymous types are supported (cf. 7.4.14, for more details), sequences or arrays need to be given a name (with **typedef**) to be used in the member declaration.

Structures may be forward declared, in particular to allow the definition of recursive structures.

Unions

IDL unions are a cross between C unions and switch statements. They may host a value of one type to be chosen between several possible cases. IDL unions must be discriminated: they embed a discriminator that indicates which case is to be used for the current instance. The possible cases as well as the type of the discriminator are part of the union declaration, whose syntax is as follows:

```
(49)   <union_dcl>            ::= <union_def>
                              |   <union_forward_dcl>
(50)   <union_def>            ::= "union" <identifier> "switch" "(" <switch_type_spec> ")"
                                  "{" <switch_body> "}"
(51)   <switch_type_spec>     ::= <integer_type>
                              |   <char_type>
                              |   <boolean_type>
                              |   <scoped_name>
(52)   <switch_body>          ::= <case>+
(53)   <case>                 ::= <case_label>+ <element_spec> ";"
(54)   <case_label>           ::= "case" <const_expr> ":"
                              |   "default" ":"
(55)   <element_spec>         ::= <type_spec> <declarator>
```

A union declaration comprises:

- The **union** keyword.
- The name given to the union (<**identifier**>).
- The type for the discriminator (<**switch_type_spec**>). That type may be either one of the following types: **integer, char, boolean** or an enumeration, or a reference (<**scoped_name**>) to one of these.
- The list of all possible cases for the union (<**switch_body**>), enclosed within braces ({}). At least one case is required. Each possibility (<**case**>) comprises the form that the union takes (<**element_spec**>) when the discriminator takes the list of specified values (<**case_label**>+). Several case labels may be associated in a single case.

- A case label must be:
    - Either a constant expression (**<const_expr>**) matching (or automatically castable) to, the defined type of the discriminator.
    - Or the **default** keyword, to tag the case when the discriminator's value does not match the other possibilities. A default case can appear at most once.
- Each possible form for the union value is made of an existing IDL type (**<type_spec>**) followed by the name given to that form (**<declarator>**).

The name of a union defines a new legal type that may be used anywhere such a type is legal in the grammar.

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:

1. If the discriminator value was explicitly listed in a case statement, the value of the element associated with that case statement.
2. If a default case label was specified, the value of the element associated with the default case label.
3. No additional value.

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

NOTE Name scoping rules require that the element declarators (all the <**declarator**> in the different <**element_spec**>) in a particular union be unique. If the <**switch_type_spec**> is an enumeration, the identifier for the enumeration is as well in the scope of the union; as a result, it must be distinct from the element declarators. The values of the constant expressions for the case labels of a single union definition must be distinct. A union type can contain a default label only where the values given in the non-default labels do not cover the entire range of the union's discriminant type.

NOTE While any ISO Latin-1 (8859-1) IDL character literal may be used in a <**case_label**> in a union definition whose discriminator type is char, not all of these characters are present in all code sets that may be used by implementation language compilers and/or runtimes, which may lead to some interoperability issue (typically leading to a **DATA_CONVERSION** system exception). Therefore, to ensure portability and interoperability, care must be exercised when assigning the <case_label> for a union member whose discriminator type is char. Due to this potential issue, use of char types as the discriminator type for unions is not recommended.

Enumerations

Enumerated types (enumerations) consist of ordered lists of enumerators. The syntax to create such a type is as follows:

```
(57)  <enum_dcl>          ::= "enum" <identifier>
                                 "{" <enumerator> { "," <enumerator> } * "}"
(58)  <enumerator>        ::= <identifier>
```

An enumeration declaration comprises:

- The **enum** keyword.
- The name given to the enumeration (<**identifier**>).
- The list of the possible values (*enumerators*) that makes the enumeration, enclosed within braces ({}). Each enumerator is identified by a specific name (<**identifier**>). In case there are several enumerators, their names are separated by commas (,). An enumeration must contain at least one enumerator and no more than 2^{32}.

The name of an enumeration defines a new legal type that may be used anywhere such a type is legal in the grammar.

NOTE The enumerated names must be mapped to a native data type capable of representing a maximally-sized enumeration. The order in which the identifiers are named in the specification of an enumeration defines the relative order of the identifiers. Any language mapping that permits two enumerators to be compared or defines successor/predecessor functions on enumerators must conform to this ordering relation.

Constructed Recursive Types and Forward Declarations

The IDL syntax allows the generation of recursive structures and unions via members that have a sequence type. For example, the following:

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

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

Forward declarations are legal for structures and unions. Their syntax is as follows:

(48)	*<struct_forward_dcl>*	*::= "struct" <identifier>*
(56)	*<union_forward_dcl>*	*::= "union" <identifier>*

A structure or union type is termed incomplete until its full definition is provided; that is, until the scope of the structure or union definition is closed by a terminating **}**;. For example:

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

If a structure or union is forward declared, a definition of that structure or union must follow the forward declaration in the same compilation unit. If this rule is violated it shall be treated as an error. Multiple forward declarations of the same structure or union are legal.

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

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

If this rule is violated, it shall be treated as an error.

Recursive definitions can span multiple levels. For example:

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

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

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

An incomplete sequence type can appear only as the element type of another sequence, or as the member type of a structure or union definition. If this rule is violated it shall be treated as an error.

7.4.1.4.4.5 Arrays

IDL defines multidimensional, fixed-size arrays. An array includes explicit sizes for each dimension. The syntax for arrays is very similar to the one of C or C++ as stated in the following rules:

(59)	*<array_declarator>*	*::= <identifier> <fixed_array_size>+*
(60)	*<fixed_array_size>*	*::= "[" <positive_int_const> "]"*

The array size (in each dimension) is fixed at compile time. The implementation of array indices is language mapping specific.

Declaring an array with all its dimensions creates an anonymous type. Within this building block, such a type cannot be used as is but needs to be given a name through a **typedef** declaration prior to any use.

7.4.1.4.4.6 Native Types

IDL provides a declaration to define an opaque type whose representation is specified by the language mapping.

As stated in the following rules, declaring a native type is done prefixing the type name (<**simple_declarator**>) with the native keyword:

(61)	*<native_dcl>*	*::= "native" <simple_declarator>*
(62)	*<simple_declarator>*	*::= <identifier>*

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

NOTE It is recommended that native types be mapped to equivalent type names in each programming language, subject to the normal mapping rules for type names in that language.

7.4.1.4.4.7 Naming Data Types

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

```
(63)  <typedef_dcl>          ::= "typedef" <type_declarator>
(64)  <type_declarator>      ::= { <simple_type_spec>
                                 | <template_type_spec>
                                 | <constr_type_dcl>
                                 } <any_declarators>
(65)  <any_declarators>      ::= <any_declarator> { "," <any_declarator> }*
```

(66)	*<any_declarator>*	*::= <simple_declarator>*
		| <array_declarator>
(59)	*<array_declarator>*	*::= <identifier> <fixed_array_size>+*
(60)	*<fixed_array_size>*	*::= "[" <positive_int_const> "]"*

Such a declaration is made of:

- The **typedef** keyword.
- The type specification, which may be a simple type specification (<**simple_type_spec**>), that is either a basic type or a scoped name denoting any IDL legal type, or a template type specification (<**template_type_spec**>), or a declaration for a constructed type (<**constr_type_dcl**>).
- A list of at least one declarator, which will provide the new type name. Each declarator can be either a simple identifier (<**simple_declarator**>), which will be then the name allocated to the type, or an array declarator (<**array_declarator**>), in which case the new name (<**identifier**> enclosed within the array declarator) will denote an array of specified type.

NOTE As previously seen, a name is also associated with a data type via the **struct, union, enum**, and **native** declarations.

NOTE Within this building block where anonymous types are forbidden, a **typedef** declaration is needed to name, prior to any use, an array or a template instantiation.

7.4.1.5 Specific Keywords

The following table selects in Table 7.6 the keywords that are specific to this building block and removes the others.

Table 7.14 — Keywords specific to Building Block Core Data Types

		boolean	case	char
		const		
	default	double		
enum			FALSE	
fixed	float			
				long
			module	
native		octet		
sequence	short	string	struct	
switch	TRUE		typedef	
		unsigned	union	
		void	wchar	wstring

7.4.2 Building Block Any

7.4.2.1 Purpose

This building block adds the ability to declare a type that may represent any valid data type.

7.4.2.2 Dependencies with other Building Blocks

This building block relies on Building Block Code Data Type.

7.4.2.3 Syntax

The following rules are added by this building block:

```
(69) <base_type_spec>          ::+<any_type>
(70)  <any_type:               ::=<any>
```

7.4.2.4 Explanations and Semantics

An **any** is a type that may represent any valid data type. At IDL level, it is just declared with the keyword **any**.

An **any** logically contains a value and some means to specify the actual type of the value. However, the specific way in which the actual type is defined is middleware-specific[5]. Each IDL language mapping provides operations that allow programmers to insert and access the value contained in an **any** as well as the actual type of that value.

7.4.2.5 Specific Keywords

The following table selects in Table 7.6 the keywords that are specific to this building block and removes the others.

Table 7.15 — Keywords specific to Building Block Any

	any			

[5] For CORBA this means is a TypeCode (see [CORBA], Part1, Sub clause 8.11 “TypeCodes”).

7.4.3 Building Block Interfaces — Basic

7.4.3.1 Purpose

This building block gathers all the rules needed to define basic interfaces, i.e., consistent groupings of operations. At this stage, there is no other implicit behavior attached to interfaces.

7.4.3.2 Dependencies with other Building Blocks

This building block relies on Building Block Core Data Types.

7.4.3.3 Syntax

The following rules form the building block:

```
(71) <definition>                    ::+<except_dcl> ";"
                                     | <interface_dcl> ";"
(72) <except_dcl>                    ::= "exception" <identifier> "{" <member>* "}"
(73) <interface_dcl>:                := <interface_def>
                                     |<interface_forward_dcl>
(74) <interface_def>                 ::= <interface_header> "{" <interface_body> "}"
(75) <interface_forward_dcl>         ::= <interface_kind> <identifier>
(76) <interface_header>              ::= <interface_kind> <identifier>
                                         [ <interface_inheritance_spec> ]
(77) <interface_kind>                ::="interface"
(78) <interface_inheritance_spec>
                                     ::= ":" <interface_name> { "," <interface_name> }*
(79) <interface_name>                ::= <scoped_name>
(80) <interface_body>                ::= <export>*
(81) <export>                        ::= <op_dcl> ";"
                                     |<attr_dcl> ";"
(82) <op_dcl>                        ::=<op_type_spec> <identifier>    "(" [ <parameter_dcls> ] ")"
                                     [ <raises_expr> ]
(83) <op_type_spec>                  ::=<type_spec>
                                     | "void"
(84) <parameter_dcls>                ::= <param_dcl> { "," <param_dcl> } *
(85) <param_dcl>                     ::=<param_attribute> <type_spec> <simple_declarator>
(86) <param_attribute>               ::= "in"
                                     | "out"
                                     | "inout"
(87) <raises_expr>                   ::= "raises" "(" <scoped_name> { "," <scoped_name> } * ")"
(88) <attr_dcl>                      ::=<readonly_attr_spec>
                                     |<attr_spec>
(89) <readonly_attr_spec>            ::="readonly" "attribute" <type_spec> <readonly_attr_declarator>
(90) <readonly_attr_declarator>
                                     ::=<simple_declarator> <raises_expr>
                                     |<simple_declarator> { "," <simple_declarator> }*
(91) <attr_spec>                     ::="attribute" <type_spec> <attr_declarator>
(92) <attr_declarator>               ::=<simple_declarator> <attr_raises_expr>
                                     |<simple_declarator> { "," <simple_declarator> }*
(93) <attr_raises_expr>              ::=<get_excep_expr> [ <set_excep_expr> ]
                                     |<set_excep_expr>
(94) <get_excep_expr>                ::="getraises" <exception_list>
(95) <set_excep_expr>                ::="setraises" <exception_list>
(96) <exception_list>                ::="(" <scoped_name> { "," <scoped_name> } * ")"
```

7.4.3.4 Explanations and Semantics

7.4.3.4.1 IDL specification

With that building block, an IDL specification may declare exceptions and interfaces (that are merely groups of operations).

```
(71)    <definition>            ::+ <except_dcl> ";"
                                 |   <interface_dcl> ";"
```

7.4.3.4.2 Exceptions

Exceptions are specific data structures, which may be returned to indicate that an exceptional condition has occurred during the execution of an operation. The syntax to declare an exception is as follows:

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

An exception declaration is made of:

- The **exception** keyword.
- An identifier (<**identifier**>) - each exception is typed with its exception identifier.
- A body enclosed within braces ({}) - the body may be void or comprise members (<**member**>*), very similar to structure members.

If an exception is returned as the outcome to an operation invocation, then the value of the exception identifier shall be accessible to the programmer for determining which particular exception was raised.

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

The way this information is made available is language-mapping specific.

An identifier declared to be an exception identifier may appear only in a **raises** expression of an operation or attribute declaration, and nowhere else.

7.4.3.4.3 Interfaces

Interfaces are groupings of elements (in this building block operations and attributes).

As defined by the following rules, an interface is made of a header (<**interface_header**>) and a body (<**interface_body**>) enclosed in braces (**{}**). An interface may also be declared with no definition using a forward declaration (<**interface_forward_dcl**>).

```
<interface_dcl>              ::=<interface_def>
                             |<interface_forward_dcl>
<interface_def>              ::=<interface_header> "{" <interface_body> "}
```

All those constructs are detailed in the following clauses.

7.4.3.4.3.1 Interface Header

An interface header is declared with the following syntax:

```
<interface_header>           ::=<interface_kind> <identifier>
                             [ <interface_inheritance_spec> ]
<interface_kind>             ::="interface"
```

An interface header comprises:

- The **interface** keyword.
- The interface name (<**identifier**>).
- Optionally an inheritance specification (<**interface_inheritance_spec**>).

The <**identifier**> that names an interface defines a new legal type. Such a type may be used anywhere a type is legal in the grammar, subject to semantic constraints as described in the following sub clauses. A parameter or structure member which is of an interface type is semantically a *reference* to an object implementing that interface. Each language binding describes how the programmer must represent such interface references.

7.4.3.4.3.2 Interface Inheritance

Interface inheritance is introduced by a colon (:) and must follow the following syntax:

```
(78)   <interface_inheritance_spec>
                                      ::= ":" <interface_name> { "," <interface_name> }*
(79)   <interface_name>              ::= <scoped_name>
```

Each <**scoped_name**> in an <**interface_inheritance_spec**> must be the name of a previously defined interface or an alias to a previously defined interface (defined using a **typedef** declaration).

Inheritance Rules

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 (in this building block, operations and attributes). In addition the elements of a base interface can be referred to as if they were elements of the derived interface.

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

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

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

The relationships between these interfaces are shown in Figure 7.1. This "diamond" shape is legal, as is the definition of E on the right.

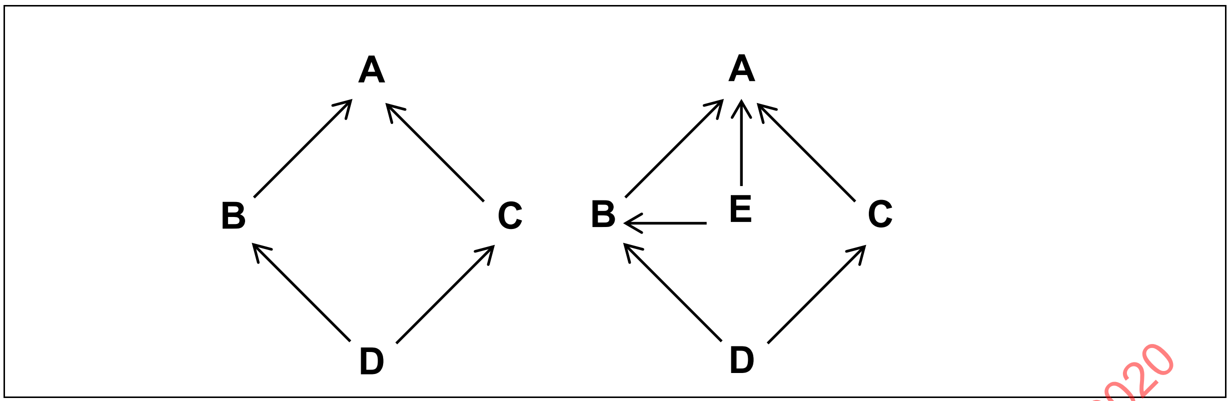

Figure 7.1 — Examples of Legal Multiple Inheritance

It is forbidden to redefine an operation or an attribute in a derived interface, as well as inheriting two different operations or attributes with the same name.

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

7.4.3.4.3.3 Interface Body

As stated in the following rules, within an interface body, operations and attributes can be defined. Those constructs are defined in the scope of the interface and exported (i.e., accessible outside the interface definition using their name scoped by the interface name).

(80)	*<interface_body>*	*::= <export>**
(81)	*<export>*	*::= <op_dcl> ";"* *| <attr_dcl> ";"*

Operations and attributes are detailed in the following sub clauses.

Operations

Operation declarations in IDL are similar to C function declarations. To define an operation, the syntax is:

(82)	*<op_dcl>*	*::= <op_type_spec> <identifier> "(" [<parameter_dcls>] ")" [<raises_expr>]*
(83)	*<op_type_spec>*	*::= <type_spec>* *| "void"*
(84)	*<parameter_dcls>*	*::= <param_dcl> { "," <param_dcl> } **
(85)	*<param_dcl>*	*::= <param_attribute> <type_spec> <simple_declarator>*
(86)	*<param_attribute>*	*::= "in"* *| "out"* *| "inout"*
(87)	*<raises_expr>*	*::= "raises" "(" <scoped_name> { "," <scoped_name> } * ")"*

An operation declaration consists of:

- The type of the operation's return result (<**op_type_spec**>); the type may be any type that can be defined in IDL. Operations that do not return a result shall specify void as return type.
- An identifier (<**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. The parameter list is enclosed between brackets (()). In case more than one parameter is declared, parameter declarations are separated by commas (,). Each parameter declaration is made of:
  - A qualifier (**<param_attribute>**) that specifies the direction in which the parameter is to be passed. The possible values are:
    - **in** - the parameter is passed from caller to operation.
    - **out** - the parameter is passed from operation to caller.
    - **inout** - the parameter is passed in both directions.
  - The type of the parameter (**<type_spec>**) that may be any valid IDL type specification.
  - The name of the parameter (**<simple_declarator>**).
- An optional expression that indicates which exceptions may be raised as a result of an invocation of this operation. This expression is made of:
  - The **raises** keyword.
  - The list of the operation-specific exceptions, enclosed between brackets (**()**) and separated by commas (**,**) in case more than one exception is specified. Each of the scoped names (**<scoped_name>**) in the list must denote a previously defined exception.

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.

In addition to any operation-specific exceptions specified in the **raises** expression, other middleware-specific standard exceptions may be raised. These exceptions are described in the related profiles.

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

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

NOTE A native type (cf. 7.4.1.4.4.6) may be used to define operation parameters, results, and exceptions. If a native type is used for an exception, it must be mapped to a type in a programming language that can be used as an exception.

Attributes

Attributes may also be declared within an interface. Declaring an attribute 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.

To create an attribute, the syntax is as follows:

```
(88)  <attr_dcl>                  ::=  <readonly_attr_spec>
                                  |    <attr_spec>
(89)  <readonly_attr_spec>        ::= "readonly" "attribute" <type_spec> <readonly_attr_declarator>
(90)  <readonly_attr_declarator>
                                  ::= <simple_declarator> <raises_expr>
                                  |   <simple_declarator> { "," <simple_declarator> }*
(91)  <attr_spec>                 ::= "attribute" <type_spec> <attr_declarator>
(92)  <attr_declarator>           ::= <simple_declarator> <attr_raises_expr>
                                  |   <simple_declarator> { "," <simple_declarator> }*
(93)  <attr_raises_expr>          ::= <get_excep_expr> [ <set_excep_expr> ]
                                  |   <set_excep_expr>
(94)  <get_excep_expr>            ::= "getraises" <exception_list>
```

(95)	***<set_excep_expr>***	***::= "setraises" <exception_list>***
(96)	***<exception_list>***	***::= "(" <scoped_name> { "," <scoped_name> } * ")"***

An attribute declaration is made of:

- An optional qualifier (**readonly**) that indicates that the attribute cannot be written. In this case, the attribute is said read-only and the declaration is equivalent to only a read accessor.
- The **attribute** keyword.
- The type of the attribute that may be any valid IDL type specification (<**type_spec**>).
- The name of the attribute (<**simple_declarator**>).
- An optional raises expression.

The optional raises expressions take different forms according to the attribute kinds:

- For read-only attributes, raises expressions are similar to those of operations (cf. rule (90) above).
- For plain attributes, raises expressions indicate which of the potential user-defined exceptions may be raised when the attribute is read (**getraises**) and which when the attribute is written (**setraises**). A plain attribute may have a **getraises** expression, a **setraises** expression or both of them. In the latter case, the **getraises** expression must be declared in first place.

The absence of a raises expression (**raises**, **getraises** or **setraises**) on an attribute implies that there are no attribute-specific exceptions. Accesses to such an attribute may still raise middleware-specific standard exceptions.

As a shortcut, several attributes can be declared in a single attribute declaration, provided that there are no attached raises clauses. In that case, the names of the attributes are listed, separated by a comma (,).

NOTE A native type (cf 7.4.1.4.4.6) may be used to define attribute types, and exceptions. If a native type is used for an exception, it must be mapped to a type in a programming language that can be used as an exception.

7.4.3.4.3.4 Forward Declaration

A forward declaration declares the name of an interface without defining it. This permits the definition of interfaces that refer to each other.

As stated in the following rule, the syntax for a forward declaration is simply the declaration of the kind of interface[6], followed by the interface name:

(75)	***<interface_forward_dcl>***	***::= <interface_kind> <identifier>***
(77)	***<interface_kind>***	***::= "interface"***

It is illegal to inherit from a forward-declared interface not previously defined.

```
module Example {
        interface base;                         // Forward declaration
        // ...
        interface derived : base {};            // Error
        interface base {};                      // Define base
        interface derived : base {};            // OK
        };
```

Multiple forward declarations of the same interface name are legal, provided that they are all consistent.

[6] The definition of a forward-declared interface must be consistent with the forward declaration: they must share the same interface kind. With this building block, there is only one interface kind (namely **interface**) however other interface-related building blocks will add other kinds (such as **abstract interface**).

7.4.3.5 Specific Keywords

The following table selects in Table 7.6 the keywords that are specific to this building block and removes the others.

Table 7.16 — Keywords specific to Building Block Interfaces — Basic

			attribute	
			exception	
		getraises		
in	inout	interface		
				out
		raises	readonly	setraises

7.4.4 Building Block Interfaces — Full

7.4.4.1 Purpose

This building block complements the former one with the ability to embed in the interface body, additional declarations such as types, exceptions and constants.

7.4.4.2 Dependencies with other Building Blocks

This building block complements Building Block Interfaces - Basic. Transitively, it relies on Building Block Core Data Types.

7.4.4.3 Syntax

The following rule is added by this building block:

```
<export>        ::+ <type_dcl> ";"
                |   <const_dcl> ";"
                |   <except_dcl> ";"
```

7.4.4.4 Explanations and Semantics

This building block adds the possibility to embed declarations of types, constants and exceptions inside an interface declaration.

The syntax for those embedded elements is exactly the same as if they were top-level constructs.

All those elements are exported (i.e., visible under the scope of their hosting interface). In contrast to attributes and operations, they may be redefined in a derived interface, which has the following consequences:

- In a derived interface, all elements of a base class may be referred to as if they were elements of the derived class, unless they are redefined in the derived class. 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. The scope rules for such names are described in 7.5.
- References to base interface elements must be unambiguous. A reference to a base interface element is ambiguous if the name is declared as a constant, type, or exception in more than one base interface. Ambiguities can be resolved by qualifying a name with its interface name (that is, using a <**scoped_name**>). It is illegal to inherit from two interfaces with the same operation or attribute name, or to redefine an operation or attribute name in the derived interface.

For example in:

```
interface A {
            typedef long L1;
            short opA (in L1 l_1);
            };
interface B {
            typedef short L1;
            L1 opB (in long l);
            };
interface C: B, A {
            typedef L1 L2;                // Error: L1 ambiguous
            typedef A::L1 L3;             // A::L1 is OK
            B::L1 opC (in L3 l_3);        // All OK no ambiguities
            };
```

- References to constants, types, and exceptions are bound to an interface when it is defined (i.e., replaced with the equivalent global scoped names). This guarantees that the syntax and semantics of an interface are not changed when the interface is a base interface for a derived interface.

Consider the following example:

```
const long L = 3;
interface A {
        typedef float coord[L]:
        void f (in coord s);          // s has three floats
        };
interface B {
        const long L = 4;
        };
interface C: B, A { };                // What is C::f()'s signature?
```

The early binding of constants, types, and exceptions at interface definition guarantees that the signature of operation **f** in interface **C** is **typedef float coord[3]; void f (in coord s);** which is identical to that in interface **A**. This rule also prevents redefinition of a constant, type, or exception in the derived interface from affecting the operations and attributes inherited from a base interface.

- Interface inheritance causes all identifiers defined in base interfaces, both direct and indirect, to be visible in the current naming scope. A type name, constant name, enumeration value name, or exception name from an enclosing scope can be redefined in the current scope. An attempt to use an ambiguous name without qualification shall be treated as an error. Thus in:

```
interface A {
        typedef string<128> string_t;
        };
interface B {
        typedef string<256> string_t;
```

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

7.4.4.5 Specific Keywords

There are no additional keywords with this building block.

7.4.5 Building Block Value Types

7.4.5.1 Purpose

This building block adds the ability to declare plain value types.

As opposed to interfaces which are merely groups of operations[7], values carry also state contents. A value type is, in some sense, half way between a "regular" IDL interface type and a structure.

Value types add the following features to the expressive power of structures:

- Single derivation (from other value types).
- Single interface support.
- Arbitrary recursive value type definitions, with sharing semantics providing the ability to define lists, trees, lattices, and more generally arbitrary graphs using value types.
- Null value semantics.

Designing a value type requires that some additional properties (state) and implementation details be specified beyond that of an interface type. Specification of this information puts some additional constraints on the implementation choices beyond that of interface types. This is reflected in both the semantics specified herein, and in the language mappings.

An essential property of value types is that their implementations are always collocated with their clients. That is, the explicit use of values in a concrete programming language is always guaranteed to use local implementations, and will not require remote calls. They have thus no system-wide identity (their value is their identity).

7.4.5.2 Dependencies with other Building Blocks

This building block requires Building Block Interfaces — Basic. Transitively, it relies on Building Block Core Data Types.

7.4.5.3 Syntax

The following rules are added by this building block:

```
(98)  <definition>              ::+<value_dcl> ";"
(99)  <value_dcl>               ::=<value_def>
                                |<value_forward_dcl>
(100) <value_def>               ::=<value_header> "{" <value_element>* "}"
(101) <value_header>            ::=       <value_kind> <identifier> [ <value_inheritance_spec> ]
(102) <value_kind>              ::=  "valuetype"
(103) <value_inheritance_spec>
                                ::= [ ":" <value_name> ] [ "supports" <interface_name> ]
(104) <value_name>              ::=<scoped_name>
(105) <value_element>           ::=<export>
                                | <state_member>
                                | <init_dcl>
```

[7] Interface attributes are actually equivalent to accessors.

```
(106) <state_member>          ::=( "public" | "private" ) <type_spec> <declarators> ";"
(107) <init_dcl>              ::="factory" <identifier> "(" [ <init_param_dcls> ] ")" [ <raises_expr> ] ";"
(108) <init_param_dcls>       ::=<init_param_dcl> { "," <init_param_dcl>}*
(109) <init_param_dcl>        ::="in" <type_spec> <simple_declarator>
(110) <value_forward_dcl>     ::=<value_kind> <identifier>
```

7.4.5.4 Explanations and Semantics

With that building block, an IDL specification may additionally declare value types, as expressed in:

```
(98)  <definition>          ::+ <value_dcl> ";"
```

There are two kinds of value type declarations: definitions of concrete (stateful) value types, and forward declarations.

```
(99)  <value_dcl>           ::= <value_def>
                            |   <value_forward_dcl>
```

7.4.5.4.1 Concrete (Stateful) Value Types

Regular value types (also named *concrete* or *stateful*) are declared with the following syntax:

```
(100) <value_def>           ::= <value_header> "{" <value_element>* "}"
```

A value declaration consists of a header (<**value_header**>) and a body, made of value elements (<**value_element**>*) enclosed within braces (**{}**). Those constructs are detailed in the following clauses.

7.4.5.4.1.1 Value Header

The value header is declared with the following syntax:

```
(101) <value_header>        ::= <value_kind> <identifier> [ <value_inheritance_spec> ]
(102) <value_kind>          ::= "valuetype"
```

The value header consists of:

- The **valuetype** keyword.
- An identifier (<**identifier**>) to name the value type. Value types may also be named by a **typedef** declaration.
- An optional value inheritance specification (<**value_inheritance_spec**>).

The name of a value type defines a new legal type that may be used anywhere such a type is legal in the grammar.

7.4.5.4.1.2 Value Inheritance Specification

As expressed in the following rules, value types may inherit from one value type and may support one interface:

```
(103) <value_inheritance_spec>
                            ::= [ ":" <value_name> ] [ "supports" <interface_name> ]
(104) <value_name>          ::= <scoped_name>
```

The value inheritance specification is thus made of two parts (both being optional):

- The inheritance from another value type, introduced by a colon sign (:) where the <**value_name**> must be the name of a previously defined value type or an alias[8] to a previously defined value type.
- The inheritance from an interface, introduced by the supports keyword, where the <**interface_name**> must be the name of a previously defined interface or an alias to a previously defined interface.

7.4.5.4.1.3 Value Element

A value can contain all the elements that an interface can (<export> in the following rule) as well as definitions of state members and initializers for that state.

(105) <value_element> ::= <export>
| <state_member>
| <init_dcl>

State Members

State members follow the following syntax:

(106) <state_member> ::= ("public" | "private") <type_spec> <declarators> ";"

Each <**state_member**> defines an element of the state, which is transmitted to the receiver when the value type is passed as a parameter. A state member is either public or private. The annotation directs the language mapping to expose or hide the different parts of the state to the clients of the value type. While its **public** part is exposed to all, the **private** part of the state is only accessible to the implementation code.

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

Initializers

In order to ensure portability of value implementations, designers may also define the signatures of initializers (or constructors) for concrete value types. Syntactically these look like local operation signatures except that they are prefixed with the keyword factory, have no return type, and must use only in parameters.

(107) <init_dcl> ::= "factory" <identifier> "(" [<init_param_dcls>] ")" [<raises_expr>] ";"

(108) <init_param_dcls> ::= <init_param_dcl> { "," <init_param_dcl>}*

(109) <init_param_dcl> ::= "in" <type_spec> <simple_declarator>

There may be any number of factory declarations. The names of the initializers are part of the name scope of the value type. Initializers defined in a value type are not inherited by derived value types, and hence the names of the initializers are free to be reused in a derived value type.

7.4.5.4.2 Forward Declarations

Similar to interfaces, value types may be forward-declared, with the following syntax:

(110) <value_forward_dcl> ::= <value_kind> <identifier>

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

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

[8] i.e., created by a typedef declaration.

It is illegal to inherit from a forward-declared value type not previously defined.

It is illegal for a value type to support a forward-declared interface not previously defined.

7.4.5.5 Specific Keywords

The following table selects in Table 7.6 the keywords that are specific to this building block and removes the others.

Table 7.17 — Keywords specific to Building Block Value Types

		factory		
	private			
public				
				supports
	valuetype			

7.4.6 Building Block CORBA-Specific — Interfaces

7.4.6.1 Purpose

This building block adds the syntactical elements that are specific to CORBA as well as provides explanations specific to a CORBA usage of the imported elements.

7.4.6.2 Dependencies with other Building Blocks

This building block is based on Building Block Interfaces - Full. Transitively, it relies on Building Block Interfaces — Basic and Building Block Core Data Types.

7.4.6.3 Syntax

This building block adds the following rules:

```
(111) <definition>          ::+<type_id_dcl> ";"
                            | <type_prefix_dcl> ";"
                            |<import_dcl> ";"
(112) <export>              ::+ <type_id_dcl> ";"
                            | <type_prefix_dcl> ";"
                            |<import_dcl> ";"
```

```
                                    | <op_oneway_dcl> ";"
                                    |<op_with_context> ";"
(113) <type_id_dcl>                 ::="typeid" <scoped_name> <string_literal>
(114) <type_prefix_dcl>             ::="typeprefix" <scoped_name> <string_literal>
(115) <import_dcl>                  ::= "import" <imported_scope>
(116) <imported_scope>              ::= <scoped_name> | <string_literal>
(117) <base_type_spec>              ::+<object_type>
(118) <object_type>                 ::="Object"
(119) <interface_kind>              ::+"local" "interface"
(120) <op_oneway_dcl>               ::="oneway" "void" <identifier> "(" [ <in_parameter_dcls> ] ")"
(121) <in_parameter_dcls>           ::=<in_param_dcl> { "," <in_param_dcl> } *
(122) <in_param_dcl>                ::="in" <type_spec> <simple_declarator>
(123) <op_with_context>             ::={<op_dcl> | <op_oneway_dcl>} <context_expr>
(124) <context_expr>                ::="context" "(" <string_literal> { "," <string_literal>* } ")"
```

7.4.6.4 Explanations and Semantics

This building block adds mainly:

- Constructs related to Interface Repository
- A named root for all interfaces (**Object**)
- Local interfaces
- One-way operations
- Operations with context
- CORBA module

All these constructs are presented in the following sub clauses as far as it is needed to understand their syntax. For more details on their precise semantics, refer to the CORBA documentation.

7.4.6.4.1 Interface Repository Related Declarations

A few new constructs related to the Interface Repository are parts of this building block:

```
(111)  <definition>           ::+ <type_id_dcl> ";"
                              |   <type_prefix_dcl> ";"
                              |   <import_dcl> ";"

(112)  <export>               ::+ <type_id_dcl> ";"
                              |   <type_prefix_dcl> ";"
                              |   <import_dcl> ";"
                              |   <op_oneway_dcl>
```

7.4.6.4.1.1 Repository Identity Declaration

The syntax of a repository identity declaration is as follows:

```
(113)  <type_id_dcl>          ::= "typeid" <scoped_name> <string_literal>
```

A repository identifier declaration includes the following elements:

- The **typeid** keyword.
- A <**scoped_name**> that denotes the named IDL construct to which the repository identifier is assigned. It must denote a previously-declared name of one of the IDL constructs that may define a scope, as explained in 7.5.2.
- A string literal (<**string_literal**>) that must contain a valid repository identifier value. This value will be assigned as the repository identity of the specified type definition.

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

7.4.6.4.1.2 Repository Identifier Prefix Declaration

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

(114)	***<type_prefix_dcl>***	***::= "typeprefix" <scoped_name> <string_literal>***

A repository identifier declaration includes the following elements:

- The **typeprefix** keyword.
- A <**scoped_name**> that denotes an IDL name scope to which the prefix applies. It must denote a previously-declared name of one of the following IDL constructs: module, interface (including abstract or local interface), value type[9] (including abstract, custom, and box value types) or event type[9] (including abstract and custom value types). A void scope ("::" as scoped name) denotes the specification scope.
- A string literal (<**string_literal**>) that must contain the string to be prefixed to repository identifiers in the specified name scope. The specified string shall be a list of one or more identifiers, separated by slashes (/). These identifiers are arbitrarily long sequences of alphabetic, digit, underscore (_), hyphen (-), and period (.) characters. The string shall not contain a trailing slash (/), and it shall not begin with the characters underscore (_), hyphen (-) or period (.).

NOTE "prefixed to the body of a repository identifier" means that the specified string is inserted into the default IDL format repository identifier immediately after the format name and colon ("**IDL**:") at the beginning of the identifier. A forward slash (/) character is inserted between the end of the specified string and the remaining body of the repository identifier.

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

7.4.6.4.1.3 Repository Id Conflict

In IDL that contains pragma prefix/ID declarations (as defined in [CORBA], Part1, Sub clause 14.7.5 "Pragma Directives for RepositoryId") and **typeprefix/typeid** declarations as explained below, both mechanisms must return the same repository id for the same IDL element otherwise an error should be raised.

Note that this rule applies only when the repository id value computation uses explicitly declared values from declarations of both kinds. If the repository id computed using explicitly declared values of one kind conflicts with one computed with implicit values of the other kind, the repository id based on explicitly declared values shall prevail.

7.4.6.4.1.4 Imports

Imports may be specified according to the following syntax:

(115)	***<import_dcl>***	***::= "import" <imported_scope>***
(116)	***<imported_scope>***	***::= <scoped_name> | <string_literal>***

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

The definition of import obviates the need to define the meaning of IDL constructs in terms of "file scopes." This standard defines the concepts of a specification as a unit of IDL expression. In the abstract, a specification consists of a finite sequence of ISO Latin-1 (8859-1) characters that form a legal IDL sentence. The physical representation of the specification is of no consequence to the definition of IDL, though it is generally associated with a file in practice.

[9] Assuming that those constructs are part of the current profile.

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

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

The effects of an import statement are as follows:

- The contents of the specified name scope are visible in the context of the importing specification. Names that occur in IDL declarations within the importing specification may be resolved to definitions in imported scopes.
- Imported IDL name scopes exist in the same space as names defined in subsequent declarations in the importing specification.
- IDL module definitions may re-open modules defined in imported name scopes.
- Importing an inner name scope (i.e., a name scope nested within one or more enclosing name scopes) does not implicitly import the contents of any of the enclosing name scopes.
- When a name scope is imported, the names of the enclosing scopes in the fully-qualified pathname of the enclosing scope are exposed within the context of the importing specification, but their contents are not imported. An importing specification may not redefine or reopen a name scope that has been exposed (but not imported) by an import statement.
- Importing a name scope recursively imports all name scopes nested within it.
- For the purposes of this International Standard, name scopes that can be imported (i.e., specified in an import statement) include the following: modules, interfaces, value types, and event types.[10]
- Redundant imports (e.g., importing an inner scope and one of its enclosing scopes in the same specification) are disregarded. The union of all imported scopes is visible to the importing program.
- This International Standard does not define a particular form for generated stubs and skeletons in any given programming language. In particular, it does not imply any normative relationship between units of specification and units of generation and/or compilation for any language mapping.

7.4.6.4.2 Object

In a CORBA scope, all interfaces inherit either directly or indirectly from a common root named **Object (CORBA::Object)**. The IDL **Object** keyword allows designating that common root in any place where an interface is allowed. This is expressed by the following additional rules:

```
(117)  <base_type_spec>        ::+ <object_type>
(118)  <object_type>           ::= "Object"
```

7.4.6.4.3 Local Interfaces

In a CORBA scope, interfaces are by default potentially remote interfaces. The keyword local allows declaring interfaces that cannot be remote.

```
(119)  <interface_kind>        ::+ "local" "interface"
```

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

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

[10] Assuming that these constructs are part of the current profile.

- A value type may support a local interface[11].
- Any IDL type, including an unconstrained interface, may appear as a parameter, attribute, return type, or exception declaration of a local interface.
- A local interface is a local type, as is any non-interface type declaration constructed using a local interface or other local type. For example, a structure, union, or exception with a member that is a local interface is also itself a local type.
- A local type may be used as a parameter, attribute, return type, or exception declaration of a local interface or of a value type.
- A local type may not appear as a parameter, attribute, return type, or exception declaration of an unconstrained interface.

See the [CORBA], Part1, Sub clause 8.3.14 "Local Object Operations" for CORBA implementation semantics associated with local objects.

7.4.6.4.4 Use of Native types

In a CORBA context, native type parameters are not permitted in operations of remote interfaces. Any attempt to transmit a value of a native type in a remote invocation may raise the MARSHAL standard system exception.

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

7.4.6.4.5 One-way Operations

By default calling applications are blocked until the called operations are complete. One-way operations are special operations that return the control to the calling applications immediately after the call. How this semantics is implemented is middleware-specific but in all cases one-way operations:

- Cannot have a result (return type must be **void**, no **out** or **inout** parameters).
- Cannot trigger exceptions.

One-way operations are declared with the following syntax:

```
(120)  <op_oneway_dcl>          ::= "oneway" "void" <identifier> "(" [ <in_parameter_dcls> ] ")"
(121)  <in_parameter_dcls>      ::= <in_param_dcl> { "," <in_param_dcl> } *
(122)  <in_param_dcl>           ::= "in" <type_spec> <simple_declarator>
```

7.4.6.4.6 Context Expressions

In a CORBA scope, operations may be added a context expression, as specified in the following additional rules:

```
(123)  <op_with_context>        ::= {<op_dcl> | <op_oneway_dcl>} <context_expr>
(124)  <context_expr>           ::= "context" "(" <string_literal> { "," <string_literal>* } ")"
```

A context expression specifies which elements of the client's context may affect the performance of a request by the object. The run-time system guarantees to make the value (if any) associated with each <**string_literal**> in the client's context available to the object implementation when the request is delivered. The ORB and/or object is free to use information in this request context during request resolution and performance.

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

[11] Assuming that those constructs are part of the current profile.

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

The mechanism by which a client associates values with the context identifiers is described in [CORBA], part 1, Sub clause 8.6 "Context Object".

7.4.6.4.7 CORBA Module

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

For example in:

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

The file **orb.idl** contains the IDL definitions for the **CORBA** module. Except for **CORBA::TypeCode**, the file **orb.idl** must be included in IDL files that use names defined in the **CORBA** module. IDL files that use **CORBA::TypeCode** may obtain its definition by including either the file **orb.idl** or the file **TypeCode.idl**.

7.4.6.5 Specific Keywords

The following table selects in Table 7.6 the keywords that are specific to this building block and removes the others.

Table 7.18 — Keywords specific to Building Block CORBA-Specific — Interfaces

				context
				import
			local	
	Object		oneway	
				typeid
	typeprefix			

7.4.7 Building Block CORBA-Specific — Value Types

7.4.7.1 Purpose

This building block adds the syntactical elements that are specific to value types when used in CORBA as well as provides explanations specific to a CORBA usage of the imported elements.

NOTE This building block has been designed as separated from Building Block CORBA-Specific — Interfaces, to allow CORBA profiles without value types.

7.4.7.2 Dependencies with other Building Blocks

This building-bock is based on Building Block Value Types and complements Building Block CORBA-Specific — Interfaces. Transitively, it relies on Building Block Interfaces — Full, Building Block Interfaces — Basic, and Building Block Core Data Types.

7.4.7.3 Syntax

This building block adds the following rules:

```
(125) <value_dcl>                    ::+ <value_box_def>
                                     |  <value_abs_def>
(126) <value_box_def>                ::= "valuetype" <identifier> <type_spec>
(127) <value_abs_def>                ::= "abstract" "valuetype" <identifier> [ <value_inheritance_spec> ]
                                      "{" <export>* "}"
(128) <value_kind>                   ::+ "custom" "valuetype"
(129) <interface_kind>               ::+ "abstract" "interface"
(130) <value_inheritance_spec>
                                     ::+ ":" ["truncatable"] <value_name> { "," <value_name> }*
                                      [ "supports" <interface_name> { "," <interface_name> }* ]
(131) <base_type_spec>               ::+ <value_base_type>
(132) <value_base_type>              ::= "ValueBase"
```

7.4.7.4 Explanations and Semantics

Main additions concern:

- Boxed value types.
- Abstract value types and interfaces as well as their impact on inheritance rules.
- Custom marshaling.
- Truncatable value types.
- ValueBase as a root for all value types.

All these constructs are presented in the following sub clauses as far as it is needed to understand their syntax. For more details on their precise semantics, refer to the CORBA documentation.

7.4.7.4.1 Boxed Value Types

It is often convenient to define a value type with no inheritance or operations and with a single state member. A shorthand IDL notation is used to simplify the use of value types for this kind of simple containment, referred to as a *value box*. Such boxed value types are defined with the following syntax:

```
(126)  <value_box_def>          ::= "valuetype" <identifier> <type_spec>
```

A boxed value type declaration simply consists of:

- The **valuetype** keyword.
- An identifier (<**identifier**>) to name the boxed value type.

- The type specification of the unique state member of the boxed value type (<**type_spec**>).

Since "boxing" a value type would add no additional properties to the value type, it is an error to box value types. Any IDL type may be used to declare a value box except a value type.

Value boxes are particularly useful for strings and sequences. Basically they avoid creating what would actually be an additional namespace that would contain only one name.

An example is the following IDL:

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

The above IDL provides similar functionality to writing the following one. However, the type identities would be different.

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

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

NOTE The declaration of a boxed value type does not open a new scope. Thus, a construction such as: **valuetype FooSeq sequence** <**FooSeq**>; is not legal IDL. The identifier being declared as a boxed value type cannot be used subsequent to its initial use and prior to the completion of the boxed value declaration.

7.4.7.4.2 Abstract Value Types and Interfaces

In this building block, value types as well as interfaces may be abstract. They are called abstract because they cannot be instantiated. Only concrete types derived from them may be actually instantiated and implemented.

Abstract types may be used to specify a type where a type specification is required (for example as a return type of an operation).

7.4.7.4.2.1 Abstract Value Types

As opposed to concrete value types, abstract value types are stateless (essentially, they are a bundle of operation signatures with a purely local implementation). To create an abstract value type, the following syntax applies:

(127) <value_abs_def> ::= "abstract" "valuetype" <identifier> [<value_inheritance_spec>] "{" <export>* "}"

No <**state_member**> or <**initializers**> may be specified. However, local operations may be specified.

Because no state information may be specified (only local operations are allowed), abstract value types are not subject to the single inheritance restrictions placed upon concrete value types. Therefore a value type may inherit from several abstract value types. In return an abstract value type cannot inherit from a concrete one.

NOTE A concrete value type with an empty state is not an abstract value type.

7.4.7.4.2.2 Abstract Interfaces

An abstract interface is an entity, which may at runtime represent either a regular interface or a value type. Like an abstract value type, it is a pure bundle of operations with no state. It is declared with specifying abstract interface as interface kind.

(129) <interface_kind> ::+ "abstract" "interface"

Unlike an abstract value type, it does not imply pass-by-value semantics, and unlike a regular interface type, it does not imply pass-by-reference semantics. Instead, the entity's runtime type determines which of these semantics are used.

An abstract interface may only inherit from abstract interfaces.

A value type may support several abstract interfaces (and only one concrete one).

7.4.7.4.3 Value Inheritance Rules

The terminology that is used to describe value type inheritance is directly analogous to that used to describe interface inheritance.

The name scoping and name collision rules for value types are identical to those for interfaces.

Values may be derived from other values and can support an interface.

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

The single immediate base concrete value type, if present, must be the first element specified in the inheritance list of the value declaration's IDL. The interfaces it supports are listed following the **supports** keyword.

While a value type may only directly support one interface, it is possible for the value type to support other interfaces as well through inheritance. In this case, the supported interface must be derived, directly or indirectly, from each interface that the value type supports through inheritance. For example:

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

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

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

These rules are summarized in the following table.

Table 7.19 — Possible inheritance relationships between value types and interfaces

May inherit from →	Abstract Interface	Interface	Abstract Value	Concrete Value	Boxed value
Abstract Interface	multiple	no	no	no	no
Interface	multiple	multiple	no	no	no
Abstract Value	supports multiple	supports single	multiple	no	no
Concrete Value	supports multiple	supports single	multiple	single	no
Boxed Value	no	no	no	no	no

7.4.7.4.4 Custom Marshaling

In a CORBA context, a value type may optionally indicate that its marshaling is custom-made by prefixing the **valuetype** keyword with **custom**, as shown in the following rule:

(128) <value_kind> ::+ "custom" "valuetype"

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

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

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

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

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

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

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

For more details regarding the implementation of custom marshaling, refer to [CORBA].

7.4.7.4.5 Truncatable

A stateful value that derives from another stateful value may specify that it is truncatable by prefixing the inheritance specification with the **truncatable** keyword as shown on the following rule:

```
(130)  <value_inheritance_spec>
                                  ::+  ":" ["truncatable"] <value_name> { "," <value_name> }*
                                       [ "supports" <interface_name> { "," <interface_name> }* ]
```

This means that the middleware is allowed to "truncate" an instance to become an instance of any of its truncatable parent (stateful) value types under certain conditions (see the CORBA documentation for more details). Note that all the intervening types in the inheritance hierarchy must be truncatable in order for truncation to a particular type to be allowed.

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

Non-custom value types may not (transitively) inherit from custom value types. Boxed value types may not be derived from, nor may they derive from, anything else.

7.4.7.4.6 Value Base

In a CORBA context, all value types have a conventional base type called **ValueBase**. This is a type, which fulfills a role that is similar to that played by **Object** for interfaces. That root may be used in an IDL specification wherever a value type may be used.

```
(131)  <base_type_spec>        ::+ <value_base_type>
(132)  <value_base_type>       ::= "ValueBase"
```

Conceptually it supports the common operations available on all value types. See Value Base Type Operation for a description of those operations. In each language mapping **ValueBase** will be mapped to an appropriate base type that supports the marshaling/unmarshaling protocol as well as the model for custom marshaling.

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

7.4.7.5 Specific Keywords

The following table selects in Table 7.6 the keywords that are specific to this building block and removes the others.

Table 7.20 — Keywords specific to Building Block CORBA-Specific — Value Types

abstract					
custom					
		truncatable			
ValueBase					

7.4.8 Building Block Components — Basic

7.4.8.1 Purpose

Purpose of that building block is to gather the minimal subset of CCM that would be useful to any component model.

7.4.8.2 Dependencies with other Building Blocks

This building block complements Building Block Interfaces - Basic. Transitively, it relies on Building Block Core Data Types.

7.4.8.3 Syntax

Thus building block adds the following rules:

```
(133) <definition>                    ::+ <component_dcl> ";"
(134) <component_dcl>                 ::= <component_def>
                                      | <component_forward_dcl>
(135) <component_forward_dcl>
                                      ::= "component" <identifier>
(136) <component_def>                 ::= <component_header> "{" <component_body> "}"
(137) <component_header>              ::= "component" <identifier> [ <component_inheritance_spec> ]
(138) <component_inheritance_spec>
                                      ::= ":" <scoped_name>
(139) <component_body>                ::=<component_export>*
(140) <component_export>              ::=<provides_dcl> ";"
                                      | <uses_dcl> ";"
                                      | <attr_dcl> ";"
(141) <provides_dcl>                  ::="provides" <interface_type> <identifier>
(142) <interface_type>                ::= <scoped_name>
(143) <uses_dcl>                      ::= "uses" <interface_type> <identifier>
```

7.4.8.4 Explanations and Semantics

This building block allows declaring simple components with basic ports.

```
(133)  <definition>              ::+ <component_dcl> ";"
```

The salient characteristics of a component declaration are as follows:

- A component declaration specifies the name of the component.
- A component may inherit from another component.
- A component declaration may include in its body any attribute declarations that are legal in normal interface declarations, together with declarations of facets and receptacles that the component defines (facets and receptacles are also called basic ports).

The syntax for declaring a component is as follows:

```
(134)  <component_dcl>           ::= <component_def>
                                 |   <component_forward_dcl>
(136)  <component_def>           ::=  <component_header> "{" <component_body> "}"
```

Basically a component definition comprises:

- A component header (<**component_header**>).
- Followed with a body (<**component_body**>) enclosed within braces ({}).

Those constructs are detailed in the following clauses.

7.4.8.4.1 Component Header

A <**component_header**> declares the primary characteristics of a component interface. Its syntax is as follows:

```
(137)  <component_header>        ::= "component" <identifier> [ <component_inheritance_spec> ]
(138)  <component_inheritance_spec>
                                 ::= ":" <scoped_name>
```

A component header comprises the following elements:

- The **component** keyword.
- An identifier (<**identifier**>) that names the component type.
- An optional inheritance specification (<**component_inheritance_spec**>), consisting of a colon (:) and a single <**scoped_name**> that must denote a previously-defined component type.

NOTE A component may inherit from at most one component. The features that are inherited by the derived component are its attributes and basic ports (facets and/or receptacles).

NOTE A component forms a naming scope, nested within the scope in which the component is declared.

7.4.8.4.2 Component Body

Its syntax is as follows:

```
(139)  <component_body>          ::= <component_export>*
(140)  <component_export>        ::= <provides_dcl> ";"
                                 |   <uses_dcl> ";"
                                 |   <attr_dcl> ";"
```

A component body can contain the following declarations:

- Facet declarations (**provides**).
- Receptacle declarations (**uses**).
- Attribute declarations (**attribute** and **readonly attribute**).

NOTE Facets and receptacles are jointly named *basic ports*.

7.4.8.4.2.1 Facets

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

A facet is declared with the following syntax:

(141) <provides_dcl> ::= "provides" <interface_type> <identifier>

(142) <interface_type> ::= <scoped_name>

A facet declaration comprises the following elements:

- The **provides** keyword.
- An <**interface_type**>, which must be a scoped name that denotes the interface type that is provided by the facet. The scoped name must denote a previously-defined non-component interface type.
- An <**identifier**> that names the facet in the scope of the component, thus allowing multiple facets of the same type to be provided by the component.

7.4.8.4.2.2 Receptacles

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

The syntax for describing a receptacle is as follows:

(143) <uses_dcl> ::= "uses" <interface_type> <identifier>

A receptacle declaration comprises the following elements:

- The **uses** keyword.
- An <**interface_type**>, which must be a scoped name that denotes the interface type that the receptacle will accept. The scoped name must denote a previously-defined non-component interface type.
- An <**identifier**> that names the receptacle in the scope of the component.

7.4.8.4.2.3 Attributes

In addition to basic ports, components may be given attributes, which are declared exactly as interface attributes.

NOTE Component attributes are intended to be used during a component instance’s initialization to establish its fundamental behavioral properties. Although the component model does not constrain the visibility or use of attributes defined on the component, it is generally assumed that they will not be of interest to the same clients that will use the component after it is configured. Rather, it is intended for use by component factories or by deployment tools in the process of instantiating an assembly of components.

7.4.8.4.3 Forward Declaration

Components may be forward-declared, which allows the definition of components that refer to each other.

As expressed in the following rule, a forward declaration consists simply of the component keyword followed by an <identifier> that names the component. The actual definition must follow later in the specification.

(135) <component_forward_dcl>
::= "component" <identifier>

Multiple forward declarations of the same component name are legal.

It is illegal to inherit from a forward-declared component not previously defined.

7.4.8.5 Specific Keywords

The following table selects in Table 7.6 the keywords that are specific to this building block and removes the others.

Table 7.21 — Keywords specific to Building Block Components — Basic

component				
				provides
				uses

7.4.9 Building Block Components — Homes

7.4.9.1 Purpose

This building block adds to the former the concept of homes. Homes are special interfaces for creating and managing instances of components.

7.4.9.2 Dependencies with Other Building Blocks

This building block complements Building Block Components - Basic. Transitively, it relies on Building Block Interfaces - Basic and Building Block Core Data Types.

7.4.9.3 Syntax

The following set of rules form the building block:

```
(144) <definition>                ::+ <home_dcl> ";"
(145) <home_dcl>                  ::= <home_header> "{" <home_body> "}"
(146) <home_header>               ::= "home" <identifier> [ <home_inheritance_spec> ]
                                        "manages" <scoped_name>
(147) <home_inheritance_spec>     ::= ":" <scoped_name>
(148) <home_body>                 ::= <home_export>*
(149) <home_export>               ::=<export>
                                  | <factory_dcl> ";"
(150) <factory_dcl>               ::= "factory" <identifier> "(" [ <factory_param_dcls> ] ")" <raises_expr> ]
(151) <factory_param_dcls>        ::=<factory_param_dcl> {"," <factory_param_dcl>}*
(152) <factory_param_dcl>         ::="in" <type_spec> <simple_declarator>
```

7.4.9.4 Explanations and Semantics

A home declaration describes an interface for managing instances of a specified component type. The salient characteristics of a home declaration are as follows:

- A home declaration must specify exactly one component type that it manages. Multiple homes may manage the same component type.
- Home declarations may include any declarations that are legal in normal interface declarations.
- Home declarations support single inheritance from other home definitions.

The syntax for a home definition is as follows:

```
(145)  <home_dcl>             ::= <home_header> "{" <home_body> "}"
```

Basically a home definition comprises:

- A home header (<**home_header**>).
- Followed with a body (<**home_body**>) enclosed within braces ({}).

Those constructs are detailed in the following clauses.

7.4.9.4.1 Home Header

A home header describes fundamental characteristics of a home interface. It is declared according to the following syntax:

```
(146)  <home_header>          ::= "home" <identifier> [ <home_inheritance_spec> ]
                                  "manages" <scoped_name>

(147)  <home_inheritance_spec>    ::=  ":" <scoped_name>
```

A home header consists of the following elements:

- The **home** keyword.
- An identifier (<**identifier**>) that names the home in the enclosing name scope.
- An optional inheritance specification (<**home_inheritance_spec**>), consisting of a colon (:) and a single scoped name that denotes a previously defined home type.
- The **manages** keyword followed by a scoped name that denotes the previously defined component type that is under the home's management.

7.4.9.4.2 Home Body

Its syntax is as follows:

(148) <home_body> ::= <home_export>*

(149) <home_export> ::= <export>
| <factory_dcl> ";"

In addition to plain operations and attributes, identical to the ones an interface body may comprise, a home body may also comprise factory operations, which are specific operations dedicated to creating component instances.

The syntax of a factory operation is as follows:

(150) <factory_dcl> ::= "factory" <identifier> "(" [<factory_param_dcls>] ")" [<raises_expr>]

(151) <factory_param_dcls> ::= <factory_param_dcl> {"," <factory_param_dcl>}*

(152) <factory_param_dcl> ::= "in" <type_spec> <simple_declarator>

A factory operation declaration consists of the following elements:

- The **factory** keyword.
- An identifier (<**identifier**>) that names the operation in the scope of the home declaration.
- An optional list of initialization parameters (<**factory_param_dcls**>) enclosed in parentheses (()). These parameters are similar to in parameters for operations. In case there are several parameters, they must be separated by a comma (,).
- An optional list of exceptions that may be raised by the operation (<**raises_expr**>).

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

7.4.9.5 Specific Keywords

The following table selects in Table 7.6 the keywords that are specific to this building block and removes the others.

Table 7.22 — Keywords specific to Building Block Components — Home

		factory		
			home	
manages				

7.4.10 Building Block CCM-Specific

7.4.10.1 Purpose

This building block complements the former one in order to support the full CCM extension to CORBA.

7.4.10.2 Dependencies with other Building Blocks

This building block relies on Building Block Components — Basic and Building Block CORBA-Specific — Value Types. Transitively, it relies on Building Block CORBA-Specific — Interfaces, Building Block Value Types, Building Block Interfaces — Full, Building Block Interfaces — Basic, and Building Block Core Data Types.

7.4.10.3 Syntax

This building block adds the following rules:

```
(153) <definition>                    ::+<event_dcl> ";"
(154) <component_header>              ::+ "component" <identifier> [ <component_inheritance_spec> ]
                                          <supported_interface_spec>
(155) <supported_interface_spec>
                                      ::= "supports" <scoped_name> { "," <scoped_name> }*
(156) <component_export>              ::+ <emits_dcl> ";"
                                      | <publishes_dcl> ";"
                                      | <consumes_dcl> ";"
(157) <interface_type>                ::+ "Object"
(158) <uses_dcl>                      ::+ "uses" "multiple" <interface_type> <identifier>
(159) <emits_dcl>                     ::= "emits" <scoped_name> <identifier>
(160) <publishes_dcl>                 ::= "publishes" <scoped_name> <identifier>
(161) <consumes_dcl>                  ::= "consumes" <scoped_name> <identifier>
(162) <home_header>                   ::+ "home" <identifier>   [ <home_inheritance_spec> ]
                                          [ <supported_interface_spec> ]
                                          "manages" <scoped_name>  [ <primary_key_spec> ]
(163) <primary_key_spec>              ::= "primarykey" <scoped_name>
(164) <home_export>                   ::+ <finder_dcl> ";"
(165) <finder_dcl>                    ::= "finder" <identifier> "(" [ <init_param_dcls> ] ")" [ <raises_expr> ]
(166) <event_dcl>                     ::= (<event_def>
                                      | <event_abs_def>
                                      | <event_forward_dcl> )
(167) <event_forward_dcl>             ::= [ "abstract" ] "eventtype" <identifier>
(168) <event_abs_def>                 ::="abstract" "eventtype" <identifier> [ <value_inheritance_spec> ]
                                         "{" <export>* "}"
(169) <event_def>                     ::= <event_header> "{" <value_element> * "}"
(170) <event_header>                  ::= [ "custom" ] "eventtype" <identifier> [ <value_inheritance_spec> ]
```

7.4.10.4 Explanations and Semantics

This building block adds mainly the following:

- Event ports.
- Finder operations in homes and keys for managed components.
- Multiple uses.
- Alignment with other CORBA specificities regarding interfaces and value types.

All these constructs are presented in the following sub clauses as far as it is needed to understand their syntax. For more details on their precise semantics, refer to [CORBA], Part3.

7.4.10.4.1 Event Support

The main addition of this building block consists in support for event interactions, namely

- The ability to define event types.
- The ability to add ports to send (publish or emit) and receive (consume) events.

The following rules express these additions:

```
(153)  <definition>            ::+ <event_dcl> ";"
(156)  <component_export>      ::+ <emits_dcl> ";"
                               |   <publishes_dcl> ";"
                               |   <consumes_dcl> ";"
```

7.4.10.4.1.1 Event Types

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

An event declaration satisfies the following syntax:

```
(166)  <event_dcl>             ::= ( <event_def>
                               |    <event_abs_def>
                               |    <event_forward_dcl> )
```

Regular Event Types

A regular event type satisfies the following syntax:

```
(169)  <event_def>             ::= <event_header> "{" <value_element> * "}"
(170)  <event_header>          ::= [ "custom" ] "eventtype" <identifier> [ <value_inheritance_spec> ]
```

The event header consists of the following elements:

- An optional modifier (**custom**) specifying whether the event type uses custom marshaling.
- The **eventype** keyword.
- The event type's name (<**identifier**>).
- An optional value inheritance specification as described in 7.4.7.4.3.

An event can contain all the elements that a value can as described in 7.4.5.4.1.3 (i.e., attributes, operations, initializers, state members).

Abstract Event Types

Event types may also be abstract. They are called abstract because an abstract event type may not be instantiated. No state members or initializers may be specified. However, local operations may be specified. Essentially they are a bundle of operation signatures with a purely local implementation.

Such an event type is declared according to the following syntax:

```
(168)  <event_abs_def>         ::= "abstract" "eventtype" <identifier> [ <value_inheritance_spec> ]
                                   "{" <export>* "}"
```

A concrete event type with an empty state is not an abstract event type.

7.4.10.4.1.2 Forward Declarations

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

(167) <event_forward_dcl> ::= ["abstract"] "eventtype" <identifier>

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

It is illegal to inherit from a forward-declared event type not previously defined.

Event Type Inheritance

As event type is a specialization of value type then event type inheritance is directly analogous to value inheritance (see 7.4.7.4.3, for a detailed description of the analogous properties for value types). In addition, an event type could inherit from a single immediate base concrete event type, which must be the first element specified in the inheritance list of the event declaration's IDL. It may be followed by other abstract values or events from which it inherits.

7.4.10.4.1.3 Event Ports

Event ports can be split into event sources and event sinks.

Event Sources — Publishers and Emitters

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

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

Publishers

The syntax for an event publisher is as follows:

(160) <publishes_dcl> ::= "publishes" <scoped_name> <identifier>

A publisher declaration consists of the following elements:

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

Emitters

The syntax for an emitter declaration is as follows:

(159) <emits_dcl> ::= "emits" <scoped_name> <identifier>

An emitter declaration consists of the following elements:

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

Event Sinks

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

The syntax for an event sink declaration is as follows:

```
(161)  <consumes_dcl>          ::= "consumes" <scoped_name> <identifier>
```

An event sink declaration contains the following elements:

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

A component may exhibit zero or more consumers.

7.4.10.4.2 Home Extensions

The second extension concerns homes.

```
(162)  <home_header>           ::+ "home" <identifier>  [ <home_inheritance_spec> ]
                                    [ <supported_interface_spec> ]
                                    "manages" <scoped_name>  [ <primary_key_spec> ]
(164)  <home_export>           ::+  <finder_dcl> ";"
```

In this profile:

- A home declaration may specify a list of interfaces that the home supports.
- A home declaration may specify a primary key type.
- *Primary keys* are values assigned by the application environment that uniquely identify component instances managed by a particular home.
- Home operations may include finder operations.
- *Finder* operations aim at retrieving components managed by the home.

7.4.10.4.2.1 Supported Interfaces

The syntax to add supported interfaces declaration within the home header is as follows:

```
(155)  <supported_interface_spec>
                               ::= "supports" <scoped_name> { "," <scoped_name> }*
```

Such a declaration consists of:

- The **supports** keyword.
- A list of <**scoped_name**>, separated by a comma (,). These scoped names must denote previously declared interfaces.

7.4.10.4.2.2 Primary Keys

The syntax for adding a primary key definition within the home header is as follows:

```
(163)  <primary_key_spec>      ::= "primarykey" <scoped_name>
```

Such a declaration consists of:

- The **primarykey** keyword.
- A scoped name (<**scoped_name**>) that denotes the primary key type. Primary key types must be value types derived from **Components::PrimaryKeyBase**. There are more specific constraints placed on primary key types, which are specified in [CORBA], Part3, "Primary key type constraints" sub clause.

7.4.10.4.2.3 Finder Operations

The syntax of a finder operation is as follows:

(165) <finder_dcl> ::= "finder" <identifier> "(" [<init_param_dcls>] ")" [<raises_expr>]

A finder operation declaration consists of the following elements:

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

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

7.4.10.4.3 Multiple Uses

The third extension consists in the ability for a receptacle to be connected to several facets. This is indicated by adding the **multiple** keyword in the receptacle definition as shown in the following rule:

(158) <uses_dcl> ::+ "uses" "multiple" <interface_type> <identifier>

The presence of this keyword indicates that the receptacle may accept multiple connections simultaneously, and results in different operations on the component's associated interface.

7.4.10.4.4 Alignment with CORBA-specific Features related to Interfaces and Value Types

7.4.10.4.4.1 Supported Interfaces in Components

As expressed in the following grammar elements, in this profile components may also support interfaces:

(154) <component_header> ::+ "component" <identifier> [<component_inheritance_spec>] <supported_interface_spec>

(155) <supported_interface_spec> ::= "supports" <scoped_name> { "," <scoped_name> }*

Within the component header such a declaration consists of:

- The **supports** keyword.
- A list of <**scoped_name**>, separated by a comma (,). These scoped names must denote previously declared interfaces.

7.4.10.4.4.2 Object Root

As for all other CORBA interfaces, in this building block, **Object** may be used wherever an interface is required. Object denotes the root for all CORBA interfaces.

(157) <interface_type> ::+ "Object"

7.4.10.5 Specific Keywords

The following table selects in Table 7.6 the keywords that are specific to this building block and removes the others.

Table 7.23 — Keywords specific to Building Block CCM — Specific

			consumes	
				emits
	eventtype			finder
				multiple
primarykey				
	publishes			

7.4.11 Building Block Components — Ports and Connectors

7.4.11.1 Purpose

This building block complements the ability to define extended ports and connectors.

7.4.11.2 Dependencies with other Building Blocks

This building block relies on the Building Block Components — Basic. Transitively, it relies on Building Block Interfaces — Basic and Building Block Core Data Types.

7.4.11.3 Syntax

The following set of rules forms the syntax of this building block:

```
(171) <definition>              ::+<porttype_dcl> ";"
                                |<connector_dcl> ";"
(172) <porttype_dcl>            ::=<porttype_def>
                                |<porttype_forward_dcl>
(173) <porttype_forward_dcl>    ::="porttype" <identifier>
(174) <porttype_def>            ::="porttype" <identifier> "{" <port_body> "}"
(175) <port_body>               ::=<port_ref> <port_export>*
(176) <port_ref>                ::=<provides_dcl> ";"
                                | <uses_dcl> ";"
                                |<port_dcl> ";"
(177) <port_export>             ::=<port_ref>
                                | <attr_dcl> ";"
(178) <port_dcl>                ::={"port" | "mirrorport"} <scoped_name> <identifier>
(179) <component_export>        ::+<port_dcl> ";"
(180) <connector_dcl>           ::=<connector_header> "{" <connector_export>+ "}"
(181) <connector_header>        ::="connector" <identifier>  [ <connector_inherit_spec> ]
```

```
(182) <connector_inherit_spec>   ::=":" <scoped_name>
(183) <connector_export>         ::=<port_ref>
                                 | <attr_dcl> ";"
```

7.4.11.4 Explanations and Semantics

As expressed in the following rule, this building block allows creating new port types (aka *extended ports*) and connectors.

```
(171)  <definition>              ::+ <porttype_dcl> ";"
                                 |   <connector_dcl> ";"
```

7.4.11.4.1 Extended Ports

An *Extended Port* is a grouping of basic ports (facets and/or receptacles) that are to be used jointly to support consistently a given interaction. Those basic ports formalize the programming contract between a component with this extended port and the connector's fragment (see below) that will realize the related interaction on behalf of the component. As such, those basic ports are always local and correspond to interfaces to be called (receptacles) or call-back interfaces (facets).

7.4.11.4.1.1 Port Type Declaration

Before it can be used in a component, an extended port has to be defined through the declaration of its type.

A port type may be defined or forward declared as expressed in the following rule:

```
(172)  <porttype_dcl>            ::= <porttype_def>
                                 |   <porttype_forward_dcl>
```

A forward declaration is made of the **porttype** keyword followed by the name of the port type (<**identifier**>). Such declarations allow attaching ports to components or connectors as well as embedding them inside other extended ports while their port types are not fully defined yet.

```
(173)  <porttype_forward_dcl>    ::= "porttype" <identifier>
```

A port type is defined with the following syntax:

```
(174)  <porttype_def>            ::= "porttype" <identifier> "{" <port_body> "}"
(175)  <port_body>               ::= <port_ref> <port_export>*
(176)  <port_ref>                ::=  <provides_dcl> ";"
                                 |   <uses_dcl> ";"
                                 |   <port_dcl> ";"
(177)  <port_export>             ::= <port_ref>
                                 |   <attr_dcl> ";"
```

Such a declaration comprises:

- The **porttype** keyword.
- An identifier that gives a name to the port type (<**identifier**>).
- A body that comprises between braces ({}):
  - At least one facet (**<provides_dcl>**) or receptacle (**<uses_dcl>**) or port declaration (**<port_dcl>**) of an already declared port type (collectively called **<port_ref>**).
  - Optionally other facets receptacles or ports as well as attributes (**<attr_dcl>**).

NOTE An extended port may thus embed another extended port. However no cycles are allowed among port type definitions.

7.4.11.4.2 Port Declaration

Once a port type has been declared, ports of that kind may be attached to components with the following syntax:

(178) <port_dcl> ::= {"port" | "mirrorport"} <scoped_name> <identifier>

(179) <component_export> ::+ <port_dcl> ";"

A port declaration comprises:

- The **port** keyword or the **mirrorport** keyword.
  Ports attached with the **port** keyword are normal extended ports. Ports attached with the **mirrorport** keyword are reverse extended ports. A reverse extended port is an extended port where all its facets are turned into receptacles, all its receptacles turned into facets, all its extended ports turned into reverse extended ports and its reverse extended ports into extended ports. Therefore an extended port and its reverse will match. Reverse extended ports may be used for components' ports although they are especially useful in connectors.
- A scoped name that identifies the port type (<**scoped_name**>). That scoped name must denote a previously declared port type.
- A name that identifies that port within the component (<**identifier**>). Several ports of the same port type may thus be attached to a single component.

7.4.11.4.3 Connectors

Connectors are used to specify interaction mechanisms between components. Connectors can have ports in the same way as components. They can be composed of simple ports (**provides** and **uses**) or extended ports (very likely in their reverse form).

Syntax to create connectors is as follows:

(180) <connector_dcl> ::= <connector_header> "{" <connector_export>+ "}"

(181) <connector_header> ::= "connector" <identifier> [<connector_inherit_spec>]

(182) <connector_inherit_spec> ::= ":" <scoped_name>

(183) <connector_export> ::= <port_ref>
| <attr_dcl> ";"

A connector declaration comprises:

- The **connector** keyword.
- An identifier which is the name of the connector (<**identifier**>).
- An optional inheritance specification made of a colon (:) followed by the name of an existing connector (<**scoped_name**>).
- A body that comprises between braces ({}):
  - At least one port (<**provides_dcl**>, <**uses_dcl**> or <**port_dcl**>).
  - Optionally other ports as well as attributes (<**attr_dcl**>).

A connector will concretely be composed of several parts (called fragments) that will consist of executors, each in charge of realizing a part of the interaction. Each fragment will be co-localized to the component using it.

7.4.11.5 Specific Keywords

The following table selects in Table 7.6 the keywords that are specific to this building block and removes the others.

Table 7.24 — Keywords specific to Building Block Components — Ports and Connectors

	connector			
		mirrorport		
		port	porttype	

7.4.12 Building Block Template Modules

7.4.12.1 Purpose

The purpose of this building block is to allow embedding constructs in *template modules*. Template modules may be parameterized by a variety of parameters (called *formal parameters*), which transitively makes all the embedded constructs parameterized by the same parameters. Before using it, a template module needs to be instantiated with values suited for the formal parameters. Instantiation of the template module instantiates all the embedded constructs.

7.4.12.2 Dependencies with other Building Blocks

Although this building block relies only on, it can be seen as orthogonal to all the other ones, meaning that all the constructs that are selected for a profile that embeds this specific building block may be embedded in a template module and thus benefit from parameterization.

7.4.12.3 Syntax

```
(184) <definition >             ::+<template_module_dcl> ";"
                                | <template_module_inst> ";"
(185) <template_module_dcl>     ::= "module" <identifier> "<" <formal_parameters> ">"
                                    "{" <tpl_definition> +"}"
(186) <formal_parameters>       ::=<formal_parameter> {"," <formal_parameter>}*
(187) <formal_parameter>        ::=<formal_parameter_type> <identifier>
(188) <formal_parameter_type>   ::= "typename" | "interface" | "valuetype" | "eventtype"
                                |  "struct" | "union" | "exception" | "enum" | "sequence"
                                |  "const" <const_type>
                                | <sequence_type>
(189) <tpl_definition>          :: =<definition>
                                | <template_module_ref> ";"
(190) <template_module_inst>    :: ="module" <scoped_name>  "<" <actual_parameters> ">" <identifier>
```

```
(191) <actual_parameters>       :: =<actual_parameter> { "," <actual_parameter>}*
(192) <actual_parameter>        ::=<type_spec>
                                | <const_expr>
(193) <template_module_ref>     :: ="alias" <scoped_name>  "<" <formal_parameter_names> ">" <identifier>
(194) <formal_parameter_names>
                                :: =<identifier> { "," <identifier>}*
```

7.4.12.4 Explanations and Semantics

This building block adds the facility to declare and instantiate template modules:

```
(184)  <definition>              ::+ <template_module_dcl> ";"
                                 |   <template_module_inst> ";"
```

7.4.12.4.1 Template Module Declaration

A template module is declared according to the following rules:

```
(185)  <template_module_dcl>    ::= "module" <identifier> "<" <formal_parameters> ">"
                                    "{" <tpl_definition> +"}"
(186)  <formal_parameters>      ::= <formal_parameter> {"," <formal_parameter>}*
(187)  <formal_parameter>       ::= <formal_parameter_type> <identifier>
(188)  <formal_parameter_type>  ::= "typename" | "interface" | "valuetype" | "eventtype"
                                 |   "struct" | "union" | "exception" | "enum" | "sequence"
                                 |   "const" <const_type>
                                 |   <sequence_type>
(189)  <tpl_definition>         ::= <definition>
                                 |   <template_module_ref> ";"
```

A template module specification comprises:

- The **module** keyword.
- An identifier for the module name (<**identifier**>).
- The specification of the formal parameters between angular brackets (< >), each of those formal parameters consisting of:
    - A type classifier (<**formal_parameter_type**>) which can be:
        - **typename** – to indicate that any valid type can be passed as parameter.
        - **Interface, valuetype, eventtype, struct, union, exception, enum, sequence** – to indicate that a more restricted type must be passed as parameter.
        - A constant type, to indicate that a constant of that type must be passed as parameter.
        - A sequence type declaration, to indicate that a compliant sequence type must be passed as parameter (the formal parameters of that sequence must appear previously in the module list of formal parameters.
    - An identifier (<**identifier**>) for the formal parameter.
- The module body (<**tpl_definition**>+), which may contain, within braces ({}) any declarations that form a classical template body (<**definition**>) as well as other template module references (<**template_module_ref**> – cf. 7.4.12.4.3). A template module cannot embed another template module. A template module cannot be re-opened (as opposed to a classical one).

7.4.12.4.2 Template Module Instantiation

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

The following rules allow template module instantiations:

```
(190)  <template_module_inst>  ::= "module" <scoped_name>  "<" <actual_parameters> ">"
                                   <identifier>
(191)  <actual_parameters>     ::= <actual_parameter> { "," <actual_parameter>}*
(192)  <actual_parameter>      ::= <type_spec>
                                   |    <const_expr>
```

A template module instantiation comprises:

- The **module** keyword.
- The name of the template module to be instantiated (<**scoped_name**>). This name must refer to a previously declared template module.
- Enclosed within angle brackets (< >), the values given to the template parameters (<**actual_parameters**>). The provided values must fit with the parameter specification as described in the previous sub clause. In particular, if the template parameter is of type "sequence type declaration," then an instantiated compliant sequence must be passed.
- The name given to the resulting module (<**identifier**>).

7.4.12.4.3 References to a Template Module

The following rules allow referencing template modules:

```
(193)  <template_module_ref>  ::= "alias" <scoped_name>  "<" <formal_parameter_names> ">"
                                  <identifier>
(194)  <formal_parameter_names>
                                  ::= <identifier> { "," <identifier>}*
```

An **alias** directive allows referencing an existing template module inside a template module definition.

This directive allows providing an alias name (which can be identical to the template module name) to the existing template module and the list of formal parameters to be used for the referenced module instantiation. Note that that list must be a subset of the formal parameters of the embedding module and that each specified formal parameter must be of a compliant type for the required one. For example:

```
module MyTemplModule <typename T, struct S, long n > {
        interface Foo {...}
        ...}

module MySecondTemplModule <typename T1, typename T2, struct S1, struct S2, long m> {
        alias MyTemplModule<T2, S2, m>   MyTemplModule;        // OK
        alias MyTemplModule2<S1, S2, m>  MyTemplModule;        // OK (S1 < T)
        alias MyTemplModule3<T2, T1, m>  MyTemplModule2;       // Error (T1 not compliant for S)

        interface Bar : MyTemplModule::Foo {...}
        ...}
```

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

7.4.12.5 Specific Keywords

The following table selects in Table 7.6 the keywords that are specific to this building block and removes the others.

Table 7.25 — Keywords specific to Building Block Template — Modules

		alias		

7.4.13 Building Block Extended Data-Types

7.4.13.1 Purpose

This building block adds a few data constructs that are proven to be useful for describing data models.

7.4.13.2 Dependencies with other Building Blocks

This building block complements the Building Block Core Data Types.

7.4.13.3 Syntax

(195) <struct_def>	**::+"struct" <identifier> ":" <scoped_name> "{" <member>* "}"**
	| "struct" <identifier> "{" "}"
(196) <switch_type_spec>	**::+<wide_char_type>**
	| <octet_type>
(197) <template_type_spec>	**::+<map_type>**
(198) <constr_type_dcl>	**::+ <bitset_dcl>**
	|<bitmask_dcl>
(199) <map_type>	**::="map" "<"<type_spec> "," <type_spec> "," <positive_int_const> ">"**
	|"map" "<" <type_spec> "," <type_spec> ">"
(200) <bitset_dcl>	**::= "bitset" <identifier> [":" <scoped_name>] "{" <bitfield>* "}"**
(201) <bitfield>	**::= <bitfield_spec> <identifier>* ";"**
(202) <bitfield_spec>	**::="bitfield" "<" <positive_int_const> ">"**
	|"bitfield" "<" <positive_int_const> "," <destination_type> ">"
(203) <destination_type>	**::= <boolean_type> | <octet_type> | <integer_type>**
(204) <bitmask_dcl>	**::= "bitmask" <identifier> "{" <bit_value> { "," <bit_value> }* "}"**
(205) <bit_value>	**::= <identifier>**
(206) <signed_int>	**::+ <signed_tiny_int>**
(207) <unsigned_int>	**::+ <unsigned_tiny_int>**

```
(208) <signed_tiny_int>          ::= "int8"
(209) <unsigned_tiny_int>        ::= "uint8"
(210) <signed_short_int>         ::+ "int16"
(211) <signed_long_int>          ::+ "int32"
(212) <signed_longlong_int>      ::+ "int64"
(213) <unsigned_short_int>       ::+ "uint16"
(214) <unsigned_long_int>        ::+ "uint32"
(215) <unsigned_longlong_int>    ::+ "uint64"
```

7.4.13.4 Explanations and Semantics

Those complements are:

- Additions to structure definition in order to support single inheritance and void content (no members).
- Ability to discriminate a union with other types (wide char and octet).
- An additional template type (maps).
- Additional constructed types (bitsets and bitmasks).

7.4.13.4.1 Structures with Single Inheritance and/or Void Content

The following rule complements the structure definition with the possibility to add a single inheritance specification and to define a structure without any member:

```
(195)  <struct_def>          ::+ "struct" <identifier> ":" <scoped_name>  "{" <member>* "}"
                             |   "struct" <identifier> "{" "}"
```

Single inheritance is denoted by a colon (:) followed by a scoped name that must correspond to the name of a previously defined structure.

When a structure type inherits from another structure type, it is considered as extending the latter, which is then considered as its base type. Members of such a structure consist in all the members of its base type plus all the ones that are declared locally.

7.4.13.4.2 Union Discriminators

In the Building Block Core Data Types, union discriminators could be the following (cf. rule (51)<union_def> ::= "union" <identifier> "switch" "(" <switch_type_spec> ")" "{" <switch_body> "}")

- Either one of the following types: **integer, char, boolean** or an **enum** type.
- Or a reference (<**scoped_name**>) to one of these.

Within this building block the following rule adds the following types: **wchar** (wide char) or **octet**

```
(196)  <switch_type_spec>    ::+ <wide_char_type>
                             |   <octet_type>
```

Accordingly, the scoped name may also reference one of these types.

7.4.13.4.3 Map, Bitset and Bitmap Types

As expressed in the following rules, the new types provided by this building-block include maps, bit sets, and bit masks:

```
(197)  <template_type_spec>  ::+ <map_type>
(198)  <constr_type_dcl>     ::+ <bitset_dcl>
                             |   <bitmask_dcl>
```

7.4.13.4.3.1 Maps

Maps are collections similar to sequences but where items are registered (and thus retrieved) associated with a key. As sequences, maps may be bounded or unbounded. As expressed in the following rule, the syntax to define map types is the same as that for sequence types with two exceptions:

- The **sequence** keyword is replaced by the new **map** keyword.
- The single type parameter that appears in a sequence definition is replaced by two type parameters in a map definition: the first one is the key element type; the second one is the value element type.

```
(199)   <map_type>        ::=   "map" "<" <type_spec> "," <type_spec> "," <positive_int_const> ">"
                          |     "map" "<" <type_spec> "," <type_spec> ">"
```

7.4.13.4.3.2 Bit Sets (including Bit Fields)

Bit sets are sequences of bits stored optimally and organized in concatenated addressable pieces called bit fields, themselves stored optimally. "Stored optimally" means that one bit uses just one bit in memory. "Concatenated" means that each bit field will be placed in memory just after its predecessor within the bit set (no alignment considerations apply).

Bit sets are similar to structures, with the following differences:

- The members of a bit set can only be bit fields
- A bit field can be anonymous, which means that it cannot be addressed. An anonymous bit field is just a placeholder to skip unused bits within a bit set.

The syntax to declare a bit set is as follows:

```
(200)   <bitset_dcl>          ::= "bitset" <identifier> [":" <scoped_name>] "{" <bitfield>* "}"
(201)   <bitfield>            ::= <bitfield_spec> <identifier>* ";"
```

Such a declaration comprises:

- The **bitset** keyword.
- The name given to the bitset (<**identifier**>).
- An optional single inheritance specification: Such a single inheritance is denoted by a colon (:) followed by a scoped name (<**scoped_name**>) that must correspond to the name of a previously defined bit set.
- The list of all bit set members (<**bitfield**>*) enclosed within braces ({}). Each member (<**bitfield**>) is defined with a specification (<**bitfield_spec**>) followed by a list of identifiers (<**identifier**>*).

Within a bit set, bit fields are sequences of bits stored optimally, to be manipulated as a whole. Their specification is as follows:

```
(202)   <bitfield_spec>       ::= "bitfield" "<" <positive_int_const> ">"
                              |   "bitfield" "<" <positive_int_const> "," <destination_type> ">"
(203)   <destination_type>    ::= <boolean_type> | <octet_type> | <integer_type>
```

It comprises:

- The **bitfield** keyword
- One or two parameters between angular brackets (< >):
    - The first one (**<positive_int_const>**) is the number of bits that can be stored (its size). The maximum value is 64.

  - The second optional one (**<destination_type>**) specifies the type that will be used to manipulate the bit field as a whole. This type can be **boolean**, **octet** or any integer type either signed or unsigned (i.e., **short**, **unsigned short**, **long**, **unsigned long**, **long long**, or **unsigned long long**).
    - When the destination type is given, the number of stored bits cannot exceed its size (i.e., 1 for **boolean**, 8 for **octet**, 16 for **short** or **unsigned short**, 32 for **long** or **unsigned long** and 64 for **long long** or **unsigned long long**).
    - When no destination type is given, it takes as default value the smallest type able to store the bit field with no loss (i.e., **boolean** if size is 1, **octet** if it is between 2 and 8, **unsigned short** if it is between 9 and 16, **unsigned long** if it is between 17 and 32 and **unsigned long long** if it is between 33 and 64).

NOTE Bit fields can only exist within a bit set.

NOTE Purpose of bit sets is to minimize as much as possible their memory footprint. In the following example, the total memory occupancy of **MyBitset** is 30:

```
bitset MyBitset {
    bitfield<3>         a;    // a is stored in 3 bits (and will be manipulable as a char)
    bitfield<1>         b;    // b is stored in 1 bit (and will be manipulable as a boolean)
    bitfield<4>;              // 4 unused bits
    bitfield<10>        c;    // c is stored in 10 bits (and will be manipulable as an unsigned short)
    bitfield<12, short> d;    // d is stored in 12 bits (and will be manipulable as a short)
    };
```

7.4.13.4.3.3 Bit Masks

Bit masks are enumerated types (like enumerations) aiming at easing bit manipulation.

A bit mask is declared with the following syntax:

```
(204)  <bitmask_dcl>     ::= "bitmask" <identifier> "{" <bit_value> { "," <bit_value> }* "}"
(205)  <bit_value>       ::= <identifier>
```

A bit mask declaration comprises:

- The **bitmask** keyword.
- The name given to the bitmask (<**identifier**>).
- The ordered list of the possible values (<**bit_value**>) that makes the bit mask, enclosed within braces ({}). Each value is identified by a specific name (<**identifier**>). In case there are several values, their names are separated by commas (,). A bit mask must contain at least one value and no more than its size expressed in bits.

By default, the size of a bit mask is 32.

Like an enumeration, a bit mask consists in a sequence of values named by an identifier. However those values are not like in a classical enumeration but computed based on their position within the bit mask, to form a mask that can be used to easily set or test the bit in that position. Those values are ordered starting with the less significant bit. For example, the actual value for the first one (which corresponds to bit in position 0) will be **0x01**, the value for the second one (position 1) **0x01 << 1** and so on as in the following example:

```
bitmask MyBitMask {
    flag0,          // 0x01 << 0
    flag1,          // 0x01 << 1
    flag2,          // 0x01 << 2
    flag3,          // 0x01 << 3
    flag4,          // 0x01 << 4
    flag5,          // 0x01 << 5
    flag6,          // 0x01 << 6
    flag7           // 0x01 << 7
    };
```

Two annotations can be used to amend a bit mask definition:

- **@bit_bound** (cf. 8.3.4.1) can annotate the whole bit mask to specify its size, which must be lower than or equal to 64. Accordingly the number of values cannot then exceed the value given to **@bit_bound**.
- **@position** (cf. 8.3.4.1) can annotate a bit value to set explicitly its position, expressed in bits, within the bit mask. Possible positions range from 0, which corresponds to the less significant bit, up to (size – 1), which corresponds to the most significant one.

The following example illustrates the use of those annotations:

```
@bit_bound(8)                                    //Actual size will be 8
bitmask MyBitMask {
    @position (0)                  flag0,        // 0x01 << 0
    @position (1)                  flag1,        // 0x01 << 1
    @position (4)                  flag4,        // 0x01 << 4
    @position (6)                  flag6,        // 0x01 << 6
    };
```

NOTE Thanks to **@position** annotations, it is possible to give values only to bits that are useful.

NOTE Although it is not recommended, annotated bit values can be declared unordered. In any cases, no duplicates are allowed.

NOTE Non-annotated bit values may be declared with annotated ones. A non-annotated bit value will always be assumed as just following the one which is before, like in the following example:

```
bitmask MyBitMask {
    @position (0)          flag0,       // 0x01 << 0
    flag1,                              // 0x01 <<1 (just after 0)
    @position (4)          flag4,       // 0x01 << 4
    @position (2)          flag2,       // 0x01 <<2
    flag3,                              // 0x01 <<3 (just after 2)
    flagx,                              // ERROR, should be 0x01 <<4  but duplicates flag4
    };
```

7.4.13.4.4 Integers restricted to holding 8-bits of information

The Building Block Core Data Types defines integer types that have well defined value ranges spanning from “short” integers that can hold 16 bits of information to “long long” integers that can hold 64 bits, see Table 7.13. It does not include an integer type restricted to holding 8 bits of information.

Within this building block the following rules add the following types: **int8** (signed 8-bit integer) and **uint8** (unsigned 8-bit integer):

(206)	***<signed_int>***	***::+ <signed_tiny_int>***
(207)	***<unsigned_int>***	***::+ <unsigned_tiny_int>***
(208)	***<signed_tiny_int>***	***::= “int8”***
(209)	***<unsigned_tiny_int>***	***::= “uint8”***

7.4.13.4.5 Explicitly-named Integer Types

The Building Block Core Data Types defines integer types using the keywords **short, long, long long, unsigned short, unsigned long, and unsigned long long**. These integer types have specified value ranges, see Table 7.13. However the value range is not explicit in the type name. This could lead to ambiguity especially for people familiar with programming languages, such as C or C++, which use the same keywords and yet don’t fully specify the value range.

Within this building block the following rules add new keywords **int16, int32, int64, uint16, uint32**, **uint64** for the same primitive types, which make the value ranges explicit:

(210) <signed_short_int> ::+ "int16"

(211) <signed_long_int> ::+ "int32"

(212) <signed_longlong_int> ::+ "int64"

(213) <unsigned_short_int> ::+ "uint16"

(214) <unsigned_long_int> ::+ "uint32"

(215) <unsigned_longlong_int> ::+ "uint64"

7.4.13.4.6 Ranges for all Integer Types

The integer types defined in the Building Block Core Data Types have their value ranges defined in see Table 7.13. The table below defines the ranges for the integer in the Building Block Extended Data-Types and their relation to the Building Block Core Data Types where appropriate.

Table 7.26 — Ranges for all integer types

Building Block Extended Data-Types Integer type	Value range	Building Block Core Data Types equivalent Integer type in (see Table 7-13)
int8	$-2^{7} \ldots 2^{7}-1$	N/A
int16	$-2^{15} \ldots 2^{15}-1$	short
int32	$-2^{31} \ldots 2^{31}-1$	long
int64	$-2^{63} \ldots 2^{63}-1$	long long
uint8	$0 \ldots 2^{8}-1$	N/A
uint16	$0 \ldots 2^{16}-1$	unsigned short
uint32	$0 \ldots 2^{32}-1$	unsigned long
uint64	$0 \ldots 2^{64}-1$	unsigned long long

7.4.13.5 Specific Keywords

The following table selects in Table 7.6 the keywords that are specific to this building block and removes the others.

Table 7.27 — Keywords specific to Building Block Extended Data-Types

				bitfield
bitmask	bitset			
	map			
int8	uint8	int16	int32	int64
uint16	uint32	uint64		

7.4.14 Building Block Anonymous Types

7.4.14.1 Purpose

The only purpose of this building block is to allow the use of anonymous types, i.e., template types or arrays that were not given a name by a **typedef** directive.

Anonymous types may cause a number of problems for language mappings and were therefore deprecated in a previous version of CORBA IDL. However, they offer an increased expression power that proves to be useful in many occasions.

The new IDL organization in building blocks allows defining profiles where anonymous types are forbidden as well as others where they will be supported:

- Profiles that do not support use of anonymous types must not embed this building block. With such a profile, all anonymous types must be given a name with a **typedef** directive before any use (see 7.4.1.4.4.7).
- Profiles that do support use of anonymous types must embed this building block.

7.4.14.2 Dependencies with other Building Blocks

This building block relies on Building Block Core Data Types.

7.4.14.3 Syntax

The two additional rules allow using anonymous types:

```
(216) <type_spec>          ::+ <template_type_spec>
(217) <declarator>         ::+ <array_declarator>
```

7.4.14.4 Explanations and Semantics

With the following rule, template types may be used at any place where a type specification is required:

```
(216)  <type_spec>                ::+ <template_type_spec>
```

NOTE A template type may be used as the type parameter for another template type. For instance, the following: **sequence<sequence<long> >** declares the type "unbounded sequence of unbounded sequence of long". For those nested template declarations, white space must be used to separate the two > tokens ending the declaration so they are not parsed as a single >> token.

With the following rule, arrays may be directly declared:

```
(217)  <declarator>               ::+ <array_declarator>
```

7.4.14.5 Specific keywords

There are no additional keywords with this building block.

7.4.15 Building Block Annotations

7.4.15.1 Purpose

This building block defines a framework to add meta-data to IDL constructs, by means of annotations. This facility, very similar to the one provided by Java, is a powerful means to extend the language without changing its syntax.

7.4.15.2 Dependencies with other Building Blocks

This building block only relies on Building Core Data Types. It is actually orthogonal to all others. This means that once defined, annotations may be applied to all the IDL constructs brought by all the building blocks that are selected to form a profile jointly with this building block.

7.4.15.3 Syntax

The following rules form the building block:

```
(218) <definition>                 ::+<annotation_dcl> " ;"
(219) <annotation_dcl>             ::=<annotation_header> "{" <annotation_body> "}"
(220) <annotation_header>          ::="@annotation" <identifier>
(221) <annotation_body>            ::= { <annotation_member>
                                   | <enum_dcl> ";"
                                   | <const_dcl> ";"
                                   | <typedef_dcl> ";" }*
(222) <annotation_member>          ::=<annotation_member_type> <simple_declarator>
                                     [ "default" <const_expr> ] ";"
(223) <annotation_member_type>
                                   ::= <const_type> | <any_const_type> | <scoped_name>
(224) <any_const_type>             ::= "any"
(225) <annotation_appl>            ::= "@" <scoped_name>  [ "(" <annotation_appl_params> ")" ]
(226) <annotation_appl_params>
                                   ::=<const_expr>
                                   |<annotation_appl_param> { "," <annotation_appl_param> }*
(227) <annotation_appl_param>
                                   ::= <identifier> "=" <const_expr>
```

7.4.15.4 Explanations and Semantics

This building block specifies how to 1) define annotations and 2) attach previously defined annotations to most IDL constructs.

7.4.15.4.1 Defining Annotations

An annotation type is a form of aggregated type similar to a structure with members that could be given constant values. An annotation is defined with a header (<**annotation_header**>) and a body (<**annotation_body**>) enclosed within braces (**{ }**):

```
(218)  <definition>           ::+ <annotation_dcl> " ;"
(219)  <annotation_dcl>       ::= <annotation_header> "{" <annotation_body> "}"
```

As expressed in the following rule:

```
(220)  <annotation_header>    ::= "@annotation" <identifier>
```

The annotation header consists of the **@annotation** keyword, followed by an identifier that is the name given to the annotation (<**identifier**>).

As expressed in the following rule:

```
(221)  <annotation_body>      ::= { <annotation_member>
                                    | <enum_dcl> ";"
                                    | <const_dcl> ";"
                                    | <typedef_dcl> ";" }*
```

An annotation body may contain:

- Annotation members (<**annotation_member**>)
- Declarations of enumeration types (<**enum_dcl**>)
- Declarations of constant values (**const_dcl**>)
- Typedef declarations (<**typedef_dcl**>)

An annotation body may be void.

As stated in the following rules:

```
(222)  <annotation_member>    ::=  <annotation_member_type> <simple_declarator>
                                  [ "default" <const_expr> ] ";"
(223)  <annotation_member_type>
                              ::= <const_type> | <any_const_type> | <scoped_name>
(224)  <any_const_type>       ::= "any"
```

An annotation member is ended by a semi-colon (;) and consists of:

- The member type (<**annotation_member_type**>), which must be a constant type (<**const_type**>) or the any keyword that means in this context any constant type[12] or a scoped name (<**scoped_name**>) which must refer to a constant type.
- The name given to the member (<**simple_declarator**>).
- An optional default value, given by a constant expression (<**const_expr**>) prefixed with the default keyword. The constant expression must be compatible with the member type.

[12] This form is useful when the annotation carries a value whose actual type depends on the element under annotation (a default value for instance).

Enumerations, constants, and typedefs declared within the annotation body may be used unscoped subsequently in the annotation body or further when applying the annotation. They are not significant anywhere else.

NOTE Annotations may be user-defined, using this syntax. They can also be implicitly defined by the IDL-processing tools. In the latter case, the behavior should be as if the related definitions were included at the beginning of the IDL specification.

NOTE Annotations should not cause more IDL-processing errors than strictly needed. Therefore, in case of multiple definitions of the same annotation in one IDL specification[13], the IDL-processing tools should accept them, provided that they are consistent. In contrast, malformed definitions shall be treated as an error.

7.4.15.4.2 Applying Annotations

An annotation, once its type is defined, may be applied with the following syntax:

```
(225)  <annotation_appl>        ::= "@" <scoped_name>  [ "(" <annotation_appl_params> ")" ]
(226)  <annotation_appl_params>
                                ::= <const_expr>
                                |   <annotation_appl_param> { "," <annotation_appl_param> }*
(227)  <annotation_appl_param>
                                ::= <identifier> "=" <const_expr>
```

Applying an annotation consists in prefixing the element under annotation with:

- The annotation name (<**scoped_name**>) prefixed with the at **symbol(@)**, also known as commercial at.
- Followed by the list of values given to the annotation's members within parentheses (()) and separated by a comma (,). Each parameter value consists of:
  - The name of the member (**<identifier>**).
  - The symbol **=**.
  - A constant expression, whose type must be compatible with the member's declaration (**<const_expr>**).

Members may appear in any order.

Members with no default value must be given a value. Members with default value may be omitted. In that case, the member is considered as valued with its default value.

Two shortened forms exist:

- In case there is no member or only one member with a default value, the annotation application may be shortened to just the name of the annotation prefixed with @**.**
- In case there is only one member, the annotation application may be shortened to the name of the annotation prefixed with @ and followed with the constant value of that unique member within (). The type of the provided constant expression must be compatible with the member's declaration.

An annotation may be applied to any IDL constructs or sub-constructs. Applying an annotation consists actually in adding the related meta-data to the element under annotation.

NOTE In case the applied annotation contains a member of type **any**, the value provided for that member must match with the element under annotation.

[13] This may happen in particular when IDL files are included.

NOTE Annotations should not cause more IDL-processing errors than strictly needed. Therefore, in case an unknown annotation is encountered, it should be ignored by the IDL-processing tools. In contrast, malformed definitions shall be treated as an error.

7.4.15.5 Specific Keywords

This building block reuses the any keyword. There is no other new keyword, if keyword is taken as "*a word, built as a valid identifier, but with a specific meaning within the language*". However, any word starting with a commercial at (**@**) will now potentially denote an annotation, starting with **@annotation** that allows creating annotation types.

Table 7.28 — Keywords specific to Building Block Annotations

	any			

7.4.16 Relationships between the Building Blocks

Even if the building blocks have been designed as independent as possible, they are linked by some dependencies. The following figure represents the graph of their relationships (actually a lattice).

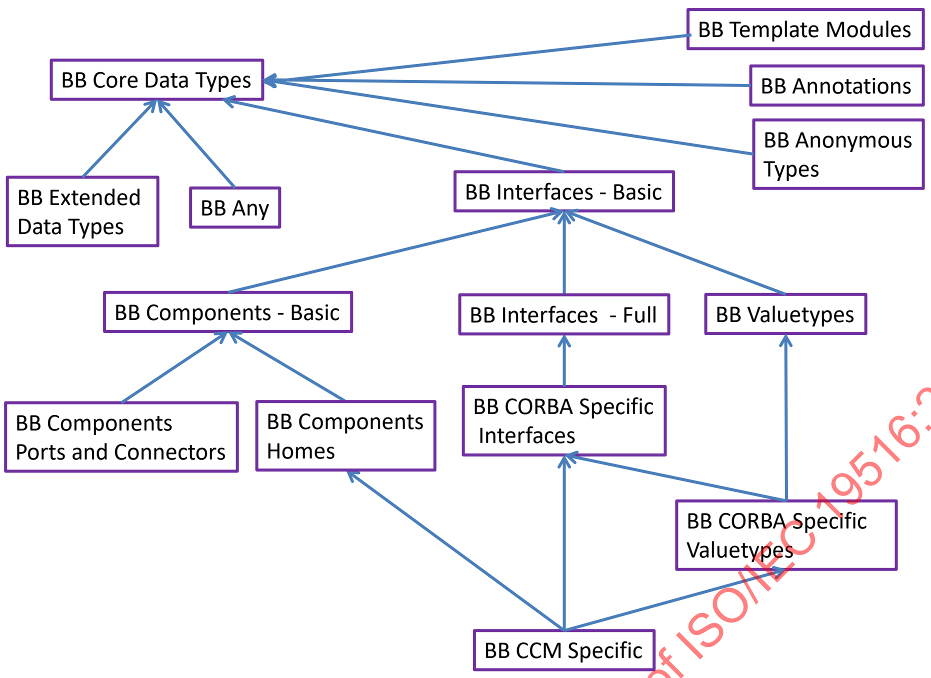

Figure 7.2 — Relationships between Building Blocks

7.5 Names and Scoping

This clause defines the visibility rules that apply to names. Those rules are considering the whole IDL grammar (i.e., the union of all building blocks). In case only a subset is used, all the considerations that apply to constructs that are not part of that subset may be simply ignored.

7.5.1 Qualified Names

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

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

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

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

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

Inheritance causes all identifiers defined in base interfaces, both direct and indirect, to be visible in derived interfaces. Such identifiers are considered to be semantically the same as the original definition. Multiple paths to the same original identifier do not conflict with each other.

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

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

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

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

```
interface A {
        typedef string<128> string_t;
        };
interface B {
        typedef string<256> string_t;
        };
interface C: A, B {
        attribute string_tTitle;            // Error: Ambiguous
        attribute A::string_t Name;         // OK
        attribute B::string_t City;         // OK
        };
```

The declaration of attribute **Title** in interface **C** is ambiguous, since the IDL-processor does not know which **string_t** is desired. Ambiguous declarations shall be treated as errors.

7.5.2 Scoping Rules and Name Resolution

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

The following kinds of definitions form scopes: modules, structures, unions, maps[14], interfaces[14], value types[14], operations[14], exceptions[14], event types[14], components[14], homes[14]. Scope applies as follows:

- The scope for a module, structure, map, interface, value type, event type, exception or home begins immediately following its opening { and ends immediately preceding its closing }.
- The scope of an operation begins immediately following its opening (and ends immediately preceding its closing).
- The scope of a union begins immediately following the (following the **switch** keyword, and ends immediately preceding its closing).

The appearance of the declaration of any of these kinds in any scope, subject to semantic validity of such declaration, opens a nested scope associated with that declaration.

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

[14] Assuming that those constructs are part of the current profile.

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

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

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

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

For example in:

```
module M {
        module Inner1 {
                typedef string S1;
                };
        module Inner2 {
                typedef string inner1;      // OK
                };
        };
```

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

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

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

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

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

```
interface A {
        enum E { E1, E2, E3 };          // line 1
        enum BadE { E3, E4, E5 };       // Error: E3 is already introduced…
                                        // …into the A scope in line 1 above
        };
interface C {
        enum AnotherE { E1, E2, E3 };
        };
interface D : C, A {
        union U switch (E) {
```