Ada Language Mapping Specification

New Edition: June 1999

Copyright 1995, 1996 BNR Europe Ltd.
Copyright 1998, Borland International
Copyright 1991, 1992, 1995, 1996 Digital Equipment Corporation
Copyright 1995, 1996 Expersoft Corporation
Copyright 1996, 1997 FUJITSU LIMITED
Copyright 1996 Genesis Development Corporation
Copyright 1989, 1990, 1991, 1992, 1995, 1996 Hewlett-Packard Company
Copyright 1991, 1992, 1995, 1996 HyperDesk Corporation
Copyright 1998 Inprise Corporation
Copyright 1996, 1997 International Business Machines Corporation
Copyright 1995, 1996 ICL, plc
Copyright 1995, 1996 IONA Technologies, Ltd.
Copyright 1996, 1997 Micro Focus Limited
Copyright 1991, 1992, 1995, 1996 NCR Corporation
Copyright 1995, 1996 Novell USG
Copyright 1991,1992, 1995, 1996 by Object Design, Inc.
Copyright 1991, 1992, 1995, 1996 Object Management Group, Inc.
Copyright 1996 Siemens Nixdorf Informationssysteme AG
Copyright 1991, 1992, 1995, 1996 Sun Microsystems, Inc.
Copyright 1995, 1996 SunSoft, Inc.
Copyright 1996 Sybase, Inc.
Copyright 1998 Telefónica Investigación y Desarrollo S.A. Unipersonal
Copyright 1996 Visual Edge Software, Ltd.

The companies listed above have granted to the Object Management Group, Inc. (OMG) a nonexclusive, royalty-free, paid up, worldwide license to copy and distribute this document and to modify this document and distribute copies of the modified version. Each of the copyright holders listed above has agreed that no person shall be deemed to have infringed the copyright in the included material of any such copyright holder by reason of having used the specification set forth herein or having conformed any computer software to the specification.

PATENT

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

NOTICE

The information contained in this document is subject to change without notice. The material in this document details an Object Management Group specification in accordance with the license and notices set forth on this page. This document does not represent a commitment to implement any portion of this specification in any company's products.

WHILE THE INFORMATION IN THIS PUBLICATION IS BELIEVED TO BE ACCURATE, THE OBJECT MANAGEMENT GROUP AND THE COMPANIES LISTED ABOVE MAKE NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS MATERIAL INCLUDING, BUT NOT LIMITED TO ANY WARRANTY OF TITLE OR OWNERSHIP, IMPLIED WARRANTY OF MERCHANTABILITY OR WARRANTY OF FITNESS FOR PARTICULAR PURPOSE OR USE. In no event shall The Object Management Group or any of the companies listed above be liable for

errors contained herein or for indirect, incidental, special, consequential, reliance or cover damages, including loss of profits, revenue, data or use, incurred by any user or any third party. The copyright holders listed above acknowledge that the Object Management Group (acting itself or through its designees) is and shall at all times be the sole entity that may authorize developers, suppliers and sellers of computer software to use certification marks, trademarks or other special designations to indicate compliance with these materials. This document contains information which is protected by copyright. All Rights Reserved. No part of this work covered by copyright herein may be reproduced or used in any form or by any means--graphic, electronic, or mechanical, including photocopying, recording, taping, or information storage and retrieval systems--without permission of the copyright owner. RESTRICTED RIGHTS LEGEND. Use, duplication, or disclosure by government is subject to restrictions as set forth in subdivision (c) (1) (ii) of the Right in Technical Data and Computer Software Clause at DFARS 252.227.7013 OMGÆ and Object Management are registered trademarks of the Object Management Group, Inc. Object Request Broker, OMG IDL, ORB, CORBA, CORBAfacilities, CORBAservices, and COSS are trademarks of the Object Management Group, Inc. X/Open is a trademark of X/Open Company Ltd.

ISSUE REPORTING

All OMG specifications are subject to continuous review and improvement. As part of this process we encourage readers to report any ambiguities, inconsistencies, or inaccuracies they may find by completing the issue reporting form at *http://www.omg.org/library/issuerpt.htm.*

Contents

Contents

Contents

Preface

0.1 About CORBA Language Mapping Specifications

The CORBA Language Mapping specifications contain language mapping information for the following languages:

- Ada
- C
- C++
- COBOL
- IDL to Java
- Java to IDL
- Smalltalk

Each language is described in a separate stand-alone volume.

0.1.1 Alignment with CORBA

The following table lists each language mapping and the version of CORBA that this language mapping is aligned with.

Language Mapping	Aligned with CORBA version
Ada	CORBA 2.0
C	CORBA 2.1
C++	CORBA 2.3
COBOL	CORBA 2.1

Language Mapping	Aligned with CORBA version
IDL to Java	CORBA 2.3
Java to IDL	CORBA 2.3
Smalltalk	CORBA 2.0

0.2 Definition of CORBA Compliance

The minimum required for a CORBA-compliant system is adherence to the specifications in CORBA Core and one mapping. Each additional language mapping is a separate, optional compliance point. Optional means users aren't required to implement these points if they are unnecessary at their site, but if implemented, they must adhere to the *CORBA* specifications to be called CORBA-compliant. For instance, if a vendor supports C++, their ORB must comply with the OMG IDL to C++ binding specified in this manual.

Interoperability and Interworking are separate compliance points. For detailed information about Interworking compliance, refer to the *Common Object Request Broker: Architecture and Specification, Interworking Architecture* chapter.

As described in the *OMA Guide*, the OMG's Core Object Model consists of a core and components. Likewise, the body of *CORBA* specifications is divided into core and component-like specifications. The structure of this manual reflects that division.

The *CORBA* specifications are divided into these volumes:

1. The *Common Object Request Broker: Architecture and Specification*, which includes the following chapters:
   - **CORBA Core**, as specified in Chapters 1-11
   - **CORBA Interoperability**, as specified in Chapters 12-16
   - **CORBA Interworking**, as specified in Chapters 17-21
2. The Language Mapping Specifications, which are organized into the following stand-alone volumes:
   - **Mapping of OMG IDL to the Ada programming language**
   - **Mapping of OMG IDL to the C programming language**
   - **Mapping of OMG IDL to the C++ programming language**
   - **Mapping of OMG IDL to the COBOL programming language**
   - **Mapping of OMG IDL to the Java programming language**
   - **Mapping of Java programming language to OMG/IDL**
   - **Mapping of OMG IDL to the Smalltalk programming language**

0.3 Acknowledgements

The following companies submitted parts of the specifications that were approved by the Object Management Group to become *CORBA* (including the Language Mapping specifications):

- BNR Europe Ltd.
- Defense Information Systems Agency
- Expersoft Corporation
- FUJITSU LIMITED
- Genesis Development Corporation
- Gensym Corporation
- IBM Corporation
- ICL plc
- Inprise Corporation
- IONA Technologies Ltd.
- Digital Equipment Corporation
- Hewlett-Packard Company
- HyperDesk Corporation
- Micro Focus Limited
- MITRE Corporation
- NCR Corporation
- Novell USG
- Object Design, Inc.
- Objective Interface Systems, Inc.
- OC Systems, Inc.
- Open Group - Open Software Foundation
- Siemens Nixdorf Informationssysteme AG
- Sun Microsystems Inc.
- SunSoft, Inc.
- Sybase, Inc.
- Telefónica Investigación y Desarrollo S.A. Unipersonal
- Visual Edge Software, Ltd.

In addition to the preceding contributors, the OMG would like to acknowledge Mark Linton at Silicon Graphics and Doug Lea at the State University of New York at Oswego for their work on the C++ mapping specification.

0.4 References

The following list of references applies to *CORBA* and/or the Language Mapping specifications:

IDL Type Extensions RFP, March 1995. OMG TC Document 95-1-35.

The Common Object Request Broker: Architecture and Specification, Revision 2.2, February 1998.

CORBAservices: Common Object Services Specification, Revised Edition, OMG TC Document 95-3-31.

COBOL Language Mapping RFP, December 1995. OMG TC document 95-12-10.

COBOL 85 ANSI X3.23-1985 / ISO 1989-1985.

IEEE Standard for Binary Floating-Point Arithmetic, ANIS/IEEE Std 754-1985.

XDR: External Data Representation Standard, RFC1832, R. Srinivasan, Sun Microsystems, August 1995.

OSF Character and Code Set Registry, OSF DCE SIG RFC 40.1 (Public Version), S. (Martin) O'Donnell, June 1994.

RPC Runtime Support For I18N Characters — Functional Specification, OSF DCE SIG RFC 41.2, M. Romagna, R. Mackey, November 1994.

X/Open System Interface Definitions, Issue 4 Version 2, 1995.

Ada Language Mapping 1

Note – The Ada Language Mapping specification is aligned with CORBA version 2.2.

The OMG document used to update this chapter was ptc/99-03-11.

Contents

This chapter contains the following sections.

1.1 Overview

The Ada language mapping provides the ability to access and implement CORBA objects in programs written in the Ada programming language (ISO/IEC 8652:1995). The mapping is based on the definition of the ORB in *Common Object Request Broker: Architecture and Specification*. The Ada language mapping uses the Ada language’s support for object oriented programming—packages, tagged types, and late binding—to present the object model described by the CORBA Architecture and Specification.

The mapping specifies how CORBA objects (objects defined by IDL) are mapped to Ada packages and types. Each CORBA object is represented by an Ada tagged type reference. The operations of mapped CORBA objects are invoked by calling primitive subprograms defined in the package associated with that object's CORBA interface.

1.1.1 Ada Implementation Requirements

The mapping is believed to map completely and correctly any legal set of definitions in the IDL language to equivalent Ada definitions. The style of this mapping is natural for Ada and does not impact the reliability either of CORBA implementations or of clients or servers built on the ORB. The mapping itself does not require any changes to CORBA.

The Ada language mapping can be implemented in a number of ways. Stub packages, ORB packages, and data types may vary between implementations of the mapping. This is a natural consequence of using an object-oriented programming language—the implementation of a package should not be visible to its user.

1.1.2 Calling Convention

Like IDL, Ada allows the passing of parameters to operations using `in`, `out`, and `in out` modes and returning values as results. The Ada language mapping preserves these in/out modes in an operation's subprogram specification. Parameters may be passed by value or by reference.

1.1.3 Memory Management

The mapping permits automatic memory management; however, the language mapping does not specify what kind, if any, of memory management facility is provided by an implementation.

1.1.4 Tasking

The mapping encourages implementors to provide tasking-safe access to CORBA services.

1.1.5 Ada Type Size Requirements

The sizes of the Ada types used to represent most IDL types are implementation dependent. That is, this mapping makes no requirements as to the 'SIZE attribute for any types except numeric types and string.

1.2 Mapping Summary

Table 1-1 summarizes the mapping of IDL constructs to Ada constructs. The following sections elaborate on each of these constructs.

Table 1-1 Summary of IDL Constructs to Ada Constructs

IDL construct	Ada construct
Source file	Library package
Module	Package (Child Package if nested)
Interface	Package with Tagged Type (Child Package if nested)
Operation	Primitive Subprogram
Attribute	"Set_*attribute*" and "*Get_attribute*" subprograms
Inheritance: Single Multiple	 Tagged Type Inheritance Tagged Type Inheritance for first parent; cover functions with explicit widening and narrowing for subsequent parents
Data types	Ada types
Exception	Exception and record type

1.2.1 Interfaces and Tagged Types

1.2.1.1 Client Side

An IDL interface is mapped to an Ada package and a tagged *reference type*. The package name will be mapped from the interface name. If the interface has an enclosing scope (including a subsystem "virtual scope"), the mapped package will be a child package of the package mapped from the enclosing scope. The mapped package will contain the definition of a tagged reference type for the object class, derived from the reference type mapped from the parent IDL interface, if the IDL interface is a subclass of another interface, or from an implementation-defined common root reference type, **`CORBA.Object.Ref`**, if the interface is not a subclass of another interface. This allows implementations of the mapping to offer automatic memory management and improves the separation of an interface and its implementation.

The mapped package also contains definitions of constants, types, exceptions, and subprograms mapped from the definitions in the interface or inherited by it.

1.2.1.2 Forward Declarations

Forward declarations result in the instantiation of a generic package that provides a reference type that can be used until the interface is fully defined. The generic instantiation also defines a nested generic package that is instantiated within the full interface definition and provides conversion from the forward reference type to the full

interface reference type and vice versa. This allows clients that hold references to the interface to convert explicitly those references to the forward reference type when required.

1.2.1.3 Server Side

The server-side mapping of an IDL interface creates a ".Impl" package that is a child of the client-side interface package. The package contains a declaration for the **`Object`** type, derived from the parent interface's object type or from a common root, **`CORBA.Object.Object`**, with a (possibly private) extension provided to allow the implementor to specify the actual data components of the object.

1.2.2 Operations

Each operation maps to an Ada subprogram with name mapped from the operation name. In the client-side package, the first (controlling) parameter to the operation is the reference type for the interface. In the server side package, the controlling parameter is a general access-to-variable type. Operations with non-void result type that have only in-mode parameters are mapped to Ada functions returning an Ada type mapped from the operation result type; otherwise, operations are mapped to Ada procedures. A non-void result is returned by an added parameter to a procedure.

1.2.3 Attributes

The Ada mapping models attributes as pairs of primitive subprograms declared in an interface package, one to set and one to get the attribute value. An attribute may be read-only, in which case only a retrieval function is provided. The name of the retrieval function is formed by prepending "**`Get_`**" to the attribute name. "**`Set_`**" is used to form the names of attribute set procedures. Like operations, a first controlling parameter is added. In client-side packages, the controlling parameter is of the reference type, while in server-side packages, it is a general access-to-variable type.

1.2.4 Inheritance

IDL inheritance allows an interface to be derived from other interfaces. IDL inheritance is interface inheritance; the only associated semantics at the IDL level are that a child object reference has "access to" all the operations of any of its parents. Reflection of IDL inheritance in mapped code is a function solely of the language mapping.

Single inheritance of IDL interfaces is directly mapped to inheritance in the Ada mapping (i.e., an interface with a parent is mapped to a tagged type that is derived from the tagged type mapped from the parent). The definitions of types, constants, and exceptions in the parent package are renamed or subtyped so that they are also "inherited" in accordance with the IDL semantics.

The client-side of multiple inheritance in IDL maps to a single Ref tagged type, as with single inheritance, where the parent type is the first interface listed in the IDL parent interface list. The IDL compiler must generate additional primitive subprograms that correspond to the operations inherited from the second and subsequent parent interfaces listed in the IDL.

1.2.5 Data Types

The mapping of types is summarized in Table 1-2.

Table 1-2 Summary of Mapping Types

Type(s)	Mapping
`Numeric`	`Corresponding Ada numeric types`
`char`	`Character`
`boolean`	`Boolean`
`octet`	`Interfaces.Unsigned_8`
`any`	`CORBA.Any (representation implementation defined)`
`struct`	`record with corresponding components`
`union`	`discriminated record`
`enum`	`enumerated type`
`sequence`	`instantiation of pre-defined generic package`
`string`	`Ada.Strings type`
`arrays`	`array types`

1.2.6 Exceptions

An IDL exception maps directly to an Ada exception declaration of the same name. The optional body of an exception maps to a type that is an extension of a predefined abstract tagged type. The components of the record will be mapped from the member of the exception body in a manner similar to the mapping of record types. Implementors must provide a function that returns the exception members from the Ada-provided **`Exception_Occurrence`** for each exception type.

1.2.7 Names and Scoping

Modules are mapped directly to packages. Nested modules map to child packages of the packages mapped from the enclosing module.

This mapping supports the introduction of a subsystem name that serves as a root virtual module for all declarations in one or more files. When specified, subsystems create a library package.

Files (actually inclusion streams) create a package to contain the “bare” definitions defined in IDL's global scope. The package name is formed from the concatenation of the file name and **_IDL_File**.

Lexical inclusion (**#include**) is mapped to with clauses for the packages mapped from the included files, modules, and interfaces.

1.3 Lexical Mapping

This section specifies the mapping of IDL identifiers, literals, and constant expressions.

1.3.1 Mapping of Identifiers

IDL identifiers follow rules similar to those of Ada but are more strict with regard to case (identifiers that differ only in case are disallowed) and less restrictive regarding the use of underscores. A conforming implementation shall map identifiers by the following rules:

- Remove any leading underscore.
- Where “_” is followed by another underscore, replace the second underscore with the character ‘U.’
- Where “_” is at the end of an identifier, add the character ’U’ after the underscore.
- When an IDL identifier collides with an Ada reserved word, insert the string “IDL_” before the identifier.

These rules cannot guarantee that name clashes will not occur. Implementations may implement additional rules to further resolve name clashes.

1.3.2 Mapping of Literals

IDL literals shall be mapped to lexically equivalent Ada literals or semantically equivalent expressions. The following sections describe the lexical mapping of IDL literals to Ada literals. This information may be used to provide semantic interpretation of the literals found in IDL constant expressions in order to calculate the value of an IDL constant or as the basis for translating those literals into equivalent Ada literals.

1.3.2.1 Integer Literals

IDL supports decimal, octal, and hexadecimal integer literals.

A decimal literal consists of a sequence of digits that does not begin with 0 (zero). Decimal literals are lexically equivalent to Ada literal values and shall be mapped "as is."

An octal literal consists of a leading ‘0’ followed by a sequence of octal digits (0 .. 7). Octal constants shall be lexically mapped by prepending “8#” and appending “#” to the IDL literal. The leading zero in the IDL literal may be deleted or kept.

A hexadecimal literal consists of “0x” or “0X” followed by a sequence of hexadecimal digits (0 .. 9, [a|A] .. [f|F]). Hexadecimal literals shall be lexically mapped to Ada literals by deleting the leading “0x” or “0X,” prepending “16#” and appending “#.”

1.3.2.2 *Floating-Point Literals*

An IDL floating-point literal consists of an integer part, a decimal point, a fraction part, an ‘e’ or ‘E,’ and an optionally signed integer exponent.

Note – IDL before version 1.2 allowed an optional type suffix [f, F, d, or D].

The integer and fraction parts consist of sequences of decimal digits. Either the integer part or the fraction part, but not both, may be missing. Either the decimal point and the fractional part or the ‘e’ (or ‘E’) and the exponent, but not both, may be missing.

A lexically equivalent floating point literal shall be formed by appending to the integer part (or “0” if the integer part is missing):

- a “.” (decimal point), the fraction part (or “0” if the fraction part is missing), or
- an “E” and the exponent (or “0” if the exponent is missing).

Optionally, the ending “E0” may be left off if the IDL did not have an exponent.

Note – For implementations choosing a mapping for the pre-1.2 optional type suffix, the following rule should be observed: If a type suffix is appended, the above construction should be appended to the Ada mapping of the type suffix followed by “’(“, and a closing “)” should be appended.

1.3.2.3 *Fixed Point Literals*

An IDL fixed-point literal consists of an integer part, a decimal point, a fraction part, and a ‘d’ or ‘D’.The integer and fraction parts consist of sequences of decimal digits. Either the integer part or the fraction part, but not both, may be missing. The decimal point may be missing if the fraction part is missing.

A lexically equivalent fixed point literal shall be formed by appending to the integer part (or “0” if the integer part is missing):

- a “.” (decimal point),
- the fraction part (or “0” if the fraction part is missing).

1.3.2.4 Character Literals

IDL character literals are single graphic characters or escape sequences enclosed by single quotes. The first form is lexically equivalent to an Ada character literal. Table 1-3 supplies lexical equivalents for the defined escape sequences. Equivalent character literals may also be used, but are not recommended when used in concatenation expressions.

Table 1-3 Lexical Equivalents for the Defined Escape Sequences

Description	IDL Escape	Octal Value	Applicable to		Ada Lexical Mapping
			char	wchar	
newline	\n	012	4	4	Ada.Characters.Latin_1.LF
horizontal tab	\t	011	4	4	Ada.Characters.Latin_1.HT
vertical tab	\v	013	4	4	Ada.Characters.Latin_1.VT
backspace	\b	010	4	4	Ada.Characters.Latin_1.BS
carriage return	\r	015	4	4	Ada.Characters.Latin_1.CR
form feed	\f	014	4	4	Ada.Characters.Latin_1.FF
alert	\a	007	4	4	Ada.Characters.Latin_1.BEL
backslash	\\	134	4	4	Ada.Characters.Latin_1.Reverse_ Solidus
question mark	\?	077	4	4	Ada.Characters.Latin_1.Question
single quote	\’	047	4	4	Ada.Characters.Latin_1.Apostrophe
double quote	\”	042	4	4	Ada.Characters.Latin_1.Quotation
octal number	\ooo	ooo	4	4	Character’val(8#ooo#)
hex number	\xhh	N/A	4		Character’val(16#hh#)
unicode character	\uhhhh	N/A		4	Wide_Character’val(16#hhhh#)

1.3.2.5 Wide Character Literals

IDL wide character literals have the form of a character literal with an “L”prefix.

1.3.2.6 String Literals

An IDL string literal is a sequence of IDL characters surrounded by double quotes. Adjacent string literals are concatenated. Within a string, the double quote character must be preceded by a ‘\’. A string literal may not contain the “nul” character. Lexically equivalent Ada string literals shall be formed as follows:

- If the string literal does not contain escape sequences (does not contain ‘\’), the IDL literal is lexically equivalent to a valid Ada literal.

- If the IDL literal contains escape sequences, the string must be partitioned into substrings. As each embedded escape sequence is encountered, three partitions must be formed:
  - one containing a substring with the contents of the string before the escape sequence,
  - one containing the escape sequence only, and
  - one containing the remainder of the string.

The remainder of the string is checked (iteratively) for additional escape sequences. The substrings containing an escape sequence must be replaced by their lexically equivalent Ada character literals as specified in the preceding section. These substrings must be concatenated together (using the Ada "&" operator) in the original order. Finally, adjacent strings must be concatenated.

1.3.2.7 Wide String Literals

Wide string literals are identical to string literals except that they are prefixed with "L". Lexically equivalent Ada wide string literals may be formed by following the above rules for strings, but substituting wide characters.

1.3.2.8 Enumeration Literals

Enumeration literals are specified by IDL identifiers. Mapping rules for enumeration literals are the same as for identifiers (see Section 1.3.1, "Mapping of Identifiers," on page 1-7).

1.3.3 Mapping of Constant Expressions

In IDL, constant expressions are used to define the values of constants in constant declarations. A subset, those expressions that evaluate to positive integer values, may also be found as:

- the maximum length of a bounded sequence,
- the maximum length of a bounded string, or as
- the fixed array size in complex declarators.

An IDL constant expression shall be mapped to an Ada static expression or a literal with the same value as the IDL constant expression. The value of the IDL expression must be interpreted according to the syntax and semantics in the *Common Object Request Broker: Architecture and Specification.* The mapping may be accomplished by interpreting the IDL constant expression yielding an equivalent Ada literal of the required type or by building an expression containing operations on literals, scoped names, and interim results that mimic the form and semantics of the IDL literal expression and yield the same value.

1.3.3.1 Mapping of Operators

Table 1-4 provides the correspondence between IDL operators in a valid constant expression and semantically equivalent Ada operators. This information may be used to provide semantic interpretation of the operators found in IDL constant expressions or as the basis for translating expressions containing those operators into equivalent Ada expressions..

Table 1-4 IDL Operators and Semantically Equivalent Ada Operators

IDL Operator	IDL symbol	Applicable Types		Ada Operator	Supported by Ada Types			
		Integer	Floating point		Boolean	Modular Integer	Signed Integer	Floating Point
or	|	√		or	√	√		
xor	^	√		xor	√	√		
and	&	√		and	√	√		
shift	<<	√		Interfaces. Shift_Left		√		
	>>	√		Interfaces. Shift_Right		√		
add	+	√	√	+		√	√	√
	-	√	√	-		√	√	√
multiply	*	√	√	*		√	√	√
	/	√	√	/		√	√	√
	%	√		rem		√	√	√
unary	-	√	√	-		√	√	√
	+	√	√	+		√	√	√
	~	√		not	√	√		
				-(value - 1)			√	

Note that the following IDL semantics (from the CORBA spec) requires some coercion of types. Differences in applicability of operators to types may force some additional type conversions to obtain Ada expressions semantically equivalent to the IDL expressions.

Mixed type expressions (e.g., integers mixed with floats) are illegal.

1.4 Mapping of Names

1.4.1 Identifiers

The lexical mapping of IDL identifiers is specified in Section 1.3.1, “Mapping of Identifiers,” on page 1-7. All identifiers in the Ada interfaces generated from IDL shall be mapped from the corresponding IDL identifiers.

1.4.2 Scoped Names

Name scopes in IDL have the following corresponding Ada named declarative regions:

- The “global” name space of IDL files are mapped to Ada “_IDL_File” library packages.
- IDL modules are mapped to Ada child packages of the packages representing their enclosing scope.
- IDL interfaces are mapped to Ada child packages of the packages representing their enclosing scope.
- All IDL constructs scoped to an interface are accessed via Ada expanded names. For example, if a type `mode` were defined in interface `printer,` then the Ada type would be referred to as `Printer.Mode.`

These mappings allow the expanded name mechanism in Ada to be used to build Ada identifiers corresponding to IDL scoped names.

1.5 Mapping of IDL Files

1.5.1 File Inclusion

While the *Common Object Request Broker: Architecture and Specification* document states that “Text in files included with a **`#include`** directive is treated as if it appeared in the including file,” a more natural Ada mapping for these includes is mapping to Ada “with clauses.” This is consistent with the primary use of the preprocessor facility which is to make available definitions from other IDL specifications and avoids the problem of redundant Ada type declarations that a literal interpretation of the inclusion would cause.

The presence of an include directive in a file shall result in Ada with clauses to library units mapped from the definition in “included” files sufficient to provide visibility (as defined by the Ada language) to all definitions referenced in included files.

Note – The simplest implementation of this requirement might be to include with clauses for all included “file packages,” module packages, interface (sub)packages, and transitively, all inclusions of the included file. However, significant readability and maintainability benefits can be gained from withing only definitions actually used.

1.5.2 Comments

The handling of comments in IDL source code is not specified; however, implementations are encouraged to transfer comment text to the generated Ada code.

1.5.3 Other Pre-Processing

Other preprocessing directives (other than #include) shall have the effect specified in the CORBA specification.

1.5.4 Global Names

The naming scope defined by an IDL file outside of any module or interface shall be mapped to an Ada package whose name shall be formed by removing the extension, if any, from the IDL source file name and appending "**`_IDL_File.`**" If all the IDL statements in a file are enclosed by a single module or interface definition, the generation of this "file package" is optional.

Note – Not generating the "file package" when not needed, permits operating system-specific file naming rules to be isolated from the resulting Ada, and so is encouraged. However, it may complicate an implementation of the withing rules for inclusion. See above.

1.6 CORBA Subsystem

The Ada mapping relies on some predefined types, packages, and functions. In the CORBA specification, these are logically defined in a module named **`CORBA`** that is automatically accessible. All Ada compilation units generated from an IDL specification shall have (non-direct) visibility to the CORBA subsystem (through a with clause.)

In the examples presented in this document, CORBA definitions may be referenced without explicit selection for simplicity. In practice, identifiers from the CORBA module would require the **`CORBA`** package prefix.

1.7 Mapping Modules

Modules define a name scope and can contain the declarations of other modules, interfaces, types, constants, and exceptions.

Top level modules (i.e., those not enclosed by other modules) shall be mapped to child packages of the subsystem package, if a subsystem is specified, or root library packages otherwise. Modules nested within other modules or within subsystems shall be mapped to child packages of the corresponding package for the enclosing module or subsystem. The name of the generated package shall be mapped from the module name.

Packages mapped from modules form an enclosing name scope for enclosed modules, interfaces, or other declarations.

Declarations scoped within an IDL module shall be mapped to declarations within the corresponding mapped Ada package.

1.8 Mapping for Interfaces (Client-Side Specific)

An IDL interface shall be mapped to a child package of the package associated with its enclosing name scope (if any) or to a root library package (if there is no enclosing name scope). This "interface package" shall define a new controlled tagged type, with name "**Ref**," used to represent object references for the mapped interface. This reference type shall be derived from an implementation-specific type named "**CORBA.Object.Ref**" or from its parent **Ref** type as specified in Section 1.8.2, "Interfaces and Inheritance," on page 1-14.

The declarations of constants, exceptions, and types scoped within interfaces shall be mapped to declarations with the mapped Ada package.

1.8.1 Object Reference Types

The use of an interface type in IDL denotes an object reference. Each IDL interface shall be mapped to an Ada controlled type. For interface **A**, the object reference type shall be named **A.Ref** (type **Ref** in Appendix A). All reference types shall be part of **CORBA.Object.Ref'CLASS** (i.e., they are derived from **CORBA.Object.Ref** or one of its descendants).

The IDL interface operations are defined as primitive operations of the Ada controlled tagged type, `Ref`. For example, if an interface defines an operation called **Op** with no parameters and **My_Ref** is a reference to the interface type, then a call would be written **A.Op(My_Ref)**.

The **Ref** controlled tagged type shall release automatically its object reference when it is deallocated, assigned a new object reference, or passes out of scope.

A reference type is a private type (i.e., its implementation is not visible to clients).

1.8.2 Interfaces and Inheritance

The reference type associated with a derived interface will inherit all of the operations of all of its parents as follows:

1. The operations of the first-named parent will be inherited through Ada's tagged type inheritance.

2. The operations of other parents shall be generated by the IDL compiler. The signature of the generated operation will be mapped as for the parent interface, but the controlling parameter shall be of the child interface reference type.

1.8.3 Mapping Forward Declarations

In IDL, a forward declaration defines the name of an interface without defining it. This allows definitions of interfaces that refer to each other. This presents a challenge to the mapping since Ada packages cannot “with” each other. An explicit mapping of forward declarations is defined in order to break this withing problem.

Conforming implementations shall provide a generic package, **`CORBA.Forward`**, with the following specification that will be used in the mapping of forward declarations.

```
with CORBA.Object;
generic
package CORBA.Forward is
   type Ref is new CORBA.Object.Ref with null record;

   generic
      type Ref_Type is new CORBA.Object.Ref with private;
   package Convert is
      function From_Forward(The_Forward : in Ref)
         return Ref_Type;
      function To_Ref      (The_Forward : in Ref)
         return Ref_Type
      renames From_Forward;
      function To_Forward (The_Ref : in Ref_Type)
         return Ref;
   end Convert;

end CORBA.Forward;
```

An instantiation of **`CORBA.Forward`** shall be performed for every forward declaration of an interface. The name of the instantiation shall be the interface name appended by “**`_Forward`**.” All references to the forward declared interface before the full declaration of the interface shall be mapped to the **`Ref`** type in this instantiated package.

Within the full declaration of the forward declared interface, the nested **`Convert`** package shall be instantiated with the actual **`Ref`** type. The name of the instantiation shall be **`Convert_Forward`**. Implementations of the contained **`To_Forward`** and **`From_Forward`** subprograms shall allow clients of the forward declaration package to convert freely from the actual **`Ref`** to the forward **`Ref`** and vice versa. Clients holding an instance of a valid reference for an interface may have to convert those references to the corresponding forward references for references mapped before the actual interface declaration.

1.8.4 Object Reference Operations

CORBA defines three operations on any object reference: duplicate, release, and is_nil. Note that these operations are on the object reference, not the object implementation. Conforming implementations shall provide these operations as follows:

- The Duplicate operation shall be provided by assignment in the Ada language.

- The other two operations shall be provided in the pre-defined package CORBA.Object (see Section 1.35, “Object,” on page 1-49) as follows:

```
-- Duplicate unneeded, use assignment

function Is_Nil(Self : Ref) return Boolean;

procedure Release(Self : in out Ref);
```

The **Is_Nil** operation returns **True** if the object reference contains an empty reference.

The **Release** procedure indicates that the caller will no longer access the reference so that associated resources may be deallocated. If the given object reference is nil, **Release** does nothing. After a call to **Release**, a call to **Is_Nil** must return **TRUE**.

1.8.5 Widening Object References

Widening of tagged types is supported by Ada through explicit type conversion and, implicitly, through parameter passing and assignment. Any object reference may be widened to the base type **CORBA.Object.Ref** using Ada syntax. Widening using Ada syntax is supported for object references in the “primary line of descent” of a particular object reference. The primary line of descent of an object reference consists of its single or first-named parent and, recursively, their single or first-named parents.

For the definitions:

```
COR      : CORBA.Object.Ref;
My_Ref : Foo.Ref;
```

the Ada language provides a natural mechanism to widen object references via view conversion:

```
COR := CORBA.Object.Ref(My_Ref);
```

An all purpose widening and narrowing method, **To_Ref**, is defined for all interfaces that provide object reference operations. This function shall support widening (and narrowing) along all lines of descent. For example, to widen an object reference to **CORBA.Object.Ref**, the **To_Ref** method defined in the **CORBA.Object** package would be used as follows:

```
function To_Ref (Self : Ref'CLASS) return Ref;
COR := CORBA.Object.To_Ref(My_Ref);
```

1.8.6 Narrowing Object References

Often it is necessary to convert an object reference from a more general type to a more specific, derived type. In particular, the root object reference IDL type Object must often be narrowed to a specific interface object reference type. Conforming

implementations must provide a **To_Ref** primitive subprogram in each interface package to perform and check the narrowing operation. Unlike widening, narrowing cannot be accomplished via normal Ada language mechanisms.

Each interface mapping shall include a function with specification:

```
function To_Ref(The_Ref : in CORBA.Object.Ref'CLASS) return
Ref;
```

The provided implementation shall be able to narrow any ancestor of the interface, regardless of whether the ancestor was defined through single or multiple inheritance. If **The_Ref** cannot be narrowed to the desired interface, this function shall raise CORBA.Bad_Param.

Narrowing an object reference can require a remote invocation (to either the target object or to an Interface Repository) to verify the relationship between the actual object and the desired narrow interface. For cases where the application programmer wishes to avoid the possibility of this remote invocation, conforming implementations must provide a primitive subprogram in each interface package to perform an unchecked narrow operation. Each interface mapping shall include a function with specification:

```
function Unchecked_To_Ref(The_Ref : in
CORBA.Object.Ref'CLASS) return Ref;
```

Regardless of whether or not **The_Ref** is known to support the desired interface, the provided implementation returns a narrowed reference.

1.8.7 Nil Object Reference

ORBs are required to define a special value of each object reference which identifies an object reference that has not been given a valid value. Conceptually, this is the "nil" value. This mapping relies on the **Is_Nil** function to detect unintialized object references, and does not require or allow definition of a Nil constant.

1.8.8 Type Object

Each occurrence of pre-defined type Object shall be mapped to **CORBA.Object.Ref.**

Type Object is a full (non-pseudo) object type. However, because it is the pre-defined root type for the Object class, its implementation does not conform to the mapping rules for interfaces and its implementation is left unspecified. See Section 1.35, "Object," on page 1-49 for more information.

1.8.9 Interface Mapping Examples

The following IDL specification:

File barn.idl

```
typedef long measure;
interface Feed {
   attribute measure weight;
};
interface Animal {
   enum State {SLEEPING, AWAKE};
   boolean eat(inout Feed bag);
   // returns true if animal is full
   attribute State alertness;
   readonly attribute Animal parent;
};
interface Horse : Animal{
   void trot(in short distance);
};
```

is mapped to these Ada packages:

```
with CORBA;

package Barn_IDL_FILE is
   type Measure is new CORBA.Long;
end Barn_IDL_FILE;

with CORBA;
with CORBA.Object;
with Barn_IDL_FILE;
package Feed is
   type Ref is new CORBA.Object.Ref with null record;
   procedure Set_Weight
      (Self : in Ref;
      To   : in Barn_IDL_FILE.Measure);
   function Get_Weight
      (Self : in Ref) return Barn_IDL_FILE.Measure;
   function To_Ref(The_Ref : in CORBA.Object.Ref'CLASS)
      return Ref;
end Feed;

with CORBA.Object;
with Feed;
package Animal is
   type Ref is new CORBA.Object.Ref with null record;
   type State is (SLEEPING, AWAKE);
procedure Eat
      (Self    : in     Ref;
      Bag     : in out Feed.Ref;
      Returns :    out Boolean);
      -- returns true if animal is full
```

```
   procedure Set_Alertness
      (Self : in Ref;
      To   : in State);
   function Get_Alertness
      (Self : in Ref) return State;
   function Get_Parent(Self : in Ref) return Ref'CLASS;
   function To_Ref(The_Ref : in CORBA.Object.Ref'CLASS)
      return Ref;
end Animal;

with Animal;

package Horse is
   type Ref is new Animal.Ref with null record;
   subtype State is Animal.State;
   procedure Trot
      (Self     : in Ref;
      Distance : in CORBA.Short);
   function To_Ref(The_Ref : in CORBA.Object.Ref'CLASS)
      return Ref;
end Horse;
```

The following illustrates the use of the forward reference mapping to resolve circular definitions. Consider the two files:

File chicken.idl:

```
#ifndef CHICKEN
#define CHICKEN
interface Chicken;
#include "egg.idl"
interface Chicken {
  Egg lay();
};
#endif
```

File egg.idl:

```
#ifndef EGG
#define EGG
interface Egg;
#include "chicken.idl"
interface Egg {
  Chicken hatch();
};
#endif
```

This use of IDL presents a difficult problem for the Ada mapping since two Ada packages cannot "with" each other. The solution is to define the operations in each interface in terms of a "forward" type; therefore, the circularity can be resolved.

```
package Chicken_IDL_FILE is

end Chicken_IDL_FILE;

with CORBA.Forward;
package Chicken_Forward is new CORBA.Forward;

with CORBA.Forward;
package Egg_Forward is new CORBA.Forward;

with CORBA.Object;
with Chicken_Forward;
with Egg_Forward;

package Egg is
   type Ref is new CORBA.Object.Ref with null record;
   function Hatch (Self : in Ref)
      return Chicken_Forward.Ref;
   function To_Ref(The_Ref : in CORBA.Object.Ref'CLASS)
      return Ref;
   package Convert is new Egg_Forward.Convert(Ref);
end Egg;

with CORBA.Object;
with Egg;
with Chicken_Forward;

package Chicken is
   type Ref is new CORBA.Object.Ref with null record;
   function Lay
      (Self : in Ref) return Egg.Ref;
   function To_Ref(The_Ref : in CORBA.Object.Ref'CLASS)
      return Ref;
   package Convert is new Chicken_Forward.Convert(Ref);
end Chicken;
```

The next example includes mapping of multiple inheritance.

This IDL:

```
interface Asset {
       ...
       void op1();
       void op2();
       ...
};
```

```
interface Vehicle {
        ...
        void op3();
        void op4();
        ...
};
interface Tank : Vehicle, Asset {
       ...
};
```

produces the following Ada code:

```
with CORBA;
package Asset is
   type Ref is new CORBA.Object.Ref with null record;

   procedure op1 (Self : Ref);
   procedure op2 (Self : Ref);

   function To_Ref (Self : CORBA.Object.Ref'CLASS)
        return Ref;
end Asset;

with CORBA;
package Vehicle is

   type Ref is new CORBA.Object.Ref with null record;

   procedure op3 (Self : Ref);
   procedure op4 (Self : Ref);

   function To_Ref (Self : CORBA.Object.Ref'CLASS)
        return Ref;
end Vehicle;

with CORBA;
with Vehicle, Asset;
package Tank is

   type Ref is new Vehicle.Ref with null record;
   function To_Ref   (Self : CORBA.Object.Ref'CLASS)
        return Ref;

   procedure op1 (Self : Ref);
   procedure op2 (Self : Ref);

end Tank;
```

1.9 Mapping for Basic Types

Several basic numeric types are defined in IDL. These types shall be mapped to Ada (sub)types. The following Ada types shall be defined in the package "CORBA" with correspondence to IDL types, as shown in Table 1-5.

Table 1-5 Ada Types with Correspondence to IDL Types

Ada Type	IDL Type	Required Range and Representation
Short	short	integer, range -(2**15) .. (2**15 - 1)
Long	long	integer, range -(2**31) .. (2**31 - 1)
Long_Long	long long	integer, range -(2**63) .. (2**63 - 1)
Unsigned_Short	unsigned short	integer, range 0 .. (2**16 - 1)
Unsigned_Long	unsigned long	integer, range 0 .. (2**32 - 1)
Unsigned_Long_Long	unsigned long long	integer, range 0 .. (2**64 - 1)
Float	float	floating point, ANSI/IEEE 754-1985 single precision
Double	double	floating point, ANSI/IEEE 754-1985 double precision
Long_Double	long double	floating point, ANSI/IEEE 754-1985 double extended precision
Char	char	8 bit ISO Latin-1 (8859.1) character set
Wchar	wchar	multi-byte character of negotiated character set
Octet	octet	integer, must include 0 .. 255

If supported, and the supported representations conform to the requirements above, the following declarations, as shown in Table 1-6, should be used.

Table 1-6 Declarations

Ada Type	Definition
CORBA.Short	type Short is new Interfaces.Integer_16;
CORBA.Long	type Long is new Interfaces.Integer_32;
CORBA.Long_Long	type Long_Long is new Interfaces.Integer_64;
CORBA.Unsigned_Short	type Unsigned_Short is new Interfaces.Unsigned_16;
CORBA.Unsigned_Long	type Unsigned_Long is new Interfaces.Unsigned_32;
CORBA.Unsigned_Long_Long	type Unsigned_Long_Long is new Interfaces.Unsigned_64;
CORBA.Float	type Float is new Interfaces.IEEE_Float_32;
CORBA.Double	type Double is new Interfaces.IEEE_Float_64;
CORBA.Long_Double	type Long_Double is new Interfaces.IEEE_Extended_Float;
CORBA.Char	subtype Char is Standard.Character;
CORBA.Wchar	subtype Wchar is Standard.Wide_Character;
CORBA.Octet	type Octet is new Interfaces.Unsigned_8;

Use of the corresponding Interfaces.C types may not meet the requirements.

1.10 Mapping for Fixed Type

The IDL fixed type shall be mapped to an equivalent Ada decimal type. The name of the mapped type shall be "**Fixed_**" prepended to the IDL specified number of digits, followed by "_", followed by the IDL specified scale factor. The corresponding Ada type definition shall have a digits value that is the same as the IDL-specified number of digits, and a delta that is a power of 10 with an exponent that is the negative value of the IDL-specified scale factor.

For example, the following IDL definition:

```
typedef fixed<8,2> Megabucks [3];
```

will map to:

```
type Fixed_8_2 is delta 0.01 digits 8;
type Megabucks is array(Integer range 0 .. 2) of Fixed_8_2;
```

1.11 Mapping for Boolean Type

The IDL boolean type shall be mapped to the CORBA Boolean type. The package CORBA will contain the definition of CORBA.Boolean as a subtype of Standard.Boolean as follows:

```
subtype Boolean is Standard.Boolean;
```

For example, the following IDL definition:

```
typedef boolean Result_Flag;
```

will map to

```
type Result_Flag is new CORBA.Boolean;
```

1.12 Mapping for Enumeration Types

An IDL **enum** type shall map directly to an Ada enumerated type with name mapped from the IDL identifier and values mapped from and in the order of the IDL member list. For example, the IDL enumeration declaration:

```
enum Color {Red, Green, Blue};
```

has the following mapping:

```
type Color is (Red, Green, Blue);
```

1.13 Mapping for Structure Types

An IDL struct type shall map directly to an Ada record type with type name mapped from the struct identifier and each component formed from each declarator in the member list as follows:

- If the declarator is a simple_declarator, the component name shall be mapped from the identifier in the declarator and the type shall be mapped from the type_spec.
- If the declarator is a complex_declarator, a preceding type definition shall define an array type. The array type name shall be mapped from the identifier contained in the array_declarator prepended to "_Array." The type definition shall be an array, over the range(s) from 0 to one less than the fixed_array_size(s) specified in the array declarator, of the type mapped from the IDL type contained in the type specification. If multiple bounds are declared, a multiple dimensional array shall be created that preserves the indexing order specified in the IDL declaration. In the component definition, the name shall be mapped from the identifier contained in the array_declarator and the type shall be the array type.

For example, the IDL struct declaration below:

```
struct Example {
   long member1, member2;
   boolean member3[4][8];
};
```

maps to the following:

```
type Member3_Array is array(0..3, 0..7) of CORBA.Boolean;
type Example is record
         Member1: CORBA.Long;
         Member2: CORBA.Long;
         Member3: Member3_Array;
      end record;
```

1.14 Mapping for Union Types

An IDL union type shall map to an Ada discriminated record type. The type name shall be mapped from the IDL identifier. The discriminant shall be formed with name "**`Switch`**" and shall be of type mapped from the IDL switch_type_spec. A default value for the discriminant shall be formed from the '**`first`** value of the mapped switch_type_spec. A variant shall be formed from each case contained in the switch_body as follows:

- Discrete_choice_list: For case_labels specified by "**`case`**" followed by a const_exp, the const_exp defines a discrete_choice. For the "**`default`**" case_label, the discrete_choice is "**`others.`**" If more than one case_label is associated with a case, they shall be "or"ed together.
- Variant component_list: The component_list of each variant shall contain one component formed from the element_spec using the mapping in Section 1.13, "Mapping for Structure Types," on page 1-24 for components.

For example, the IDL union declaration below:

```
union Example switch (long) {
   case 1: case 3: long Counter;
   case 2: boolean Flags [4] [8];
   default: long Unknown;
};
```

maps to the following:

```
type Flags_Array is array( 0..3, 0.. 7) of Boolean;
type Example(Switch : CORBA.Long := CORBA.Long'first) is
record
   case Switch is
      when 1 | 3 =>
         Counter: CORBA.Long;
      when 2 =>
         Flags: Flags_Array;
      when others =>
         Unknown : CORBA.Long;
   end case;
end record;
```

1.15 Mapping for Sequence Types

IDL defines a sequence as a "one-dimensional array with two characteristics: a maximum size (which is fixed at compile time) and a length (which is determined at run time)." The syntax is:

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

Note that a simple_type_spec can include any of the basic IDL types, any scoped name, or any template type. Thus, sequences can also be anonymously defined within a nested sequence declaration. A sequence type specification can also be contained in a typedef, in a declaration of a struct member, or in a definition of a union case.

A sequence is mapped to an Ada type that behaves similarly to an unconstrained array.

Two Ada generic package specifications, **`CORBA.Sequences.Bounded`** and **`CORBA.Sequences.Unbounded`** define the interface to the sequence type operations. Conforming implementation of the packages defining the sequence types shall provide value semantics for assignment (as opposed to reference semantics).

Thus, the implementation of assignment of one sequence variable to another sequence variable must first destroy the memory of the target sequence variable and then perform a deep-copy of the second sequence variable to the target sequence variable.

Each sequence type declaration shall correspond to an instantiation of **`CORBA.Sequences.Bounded`** or **`CORBA.Sequences.Unbounded`**, as appropriate. The formal of the generic packages and the actual arguments provided are implementation defined. The name and scope of the instantiation is left implementation defined.

The following sequence types in DrawingKit:

IDL File: drawing.idl

```
module Fresco {
interface DrawingKit {
   typedef sequence<octet> Data8;
   typedef sequence<long, 1024>  Data32;
};
};
```

map to generic package instantiations, as follows:

```
package Fresco is
end Fresco;

with CORBA.Sequences;
with CORBA.Object;

package Fresco.DrawingKit is

   type Ref is new CORBA.Object.Ref with null record;
   package IDL_SEQUENCE_octet is
      new CORBA.Sequences.Unbounded
         (CORBA.Octet);
   type Data8 is new IDL_SEQUENCE_octet.Sequence;

   package IDL_SEQUENCE_1024_long is
      new CORBA.Sequences.Bounded
         (CORBA.Long, 1024);
   type Data32 is new IDL_SEQUENCE_1024_long.Sequence;

end Fresco.DrawingKit;
```

Note that for the purposes of other rules, the “type mapped from” a sequence declaration is the “.Sequence” type of the instantiated package. This is relevant to the rules for Typedefs (“Mapping for Typedefs” on page 1-30) and for other template types. Thus, in the previous example, the instantiated “**`.Sequence`**” type is followed by a type derivation. Also, the following declaration:

typedef sequence<sequence<octet>> Ragged8;

will map to

```
with CORBA.Sequences.Unbounded;
...
package IDL_SEQUENCE_octet is
   CORBA.Sequences.Unbounded(CORBA.Octet);

package IDL_SEQUENCE_SEQUENCE_octet is
   new CORBA.Sequences.Unbounded
      (IDL_SEQUENCE_octet.Sequence);

type Ragged8 is new IDL_SEQUENCE_SEQUENCE_octet.Sequence
```

1.16 Mapping for String Types

The IDL bounded and unbounded strings types are mapped to Ada's predefined string packages or functional equivalent.

Conforming implementations shall provide an unbounded string type in the package **CORBA**. The **CORBA.String** type shall be a derivation of **Ada.Strings.Unbounded.Unbounded_String** or a functionally equivalent package with equivalent primitive operations. Conforming implementations shall define a **CORBA.Null_String** constant. In addition to the subprograms provided by **Ada.Strings.Unbounded**, conforming implementations shall provide the following additional functions in package **CORBA**:

```
function To_CORBA_String (Source : Standard.String)
   return CORBA.String;
function To_Standard_String (Source : CORBA.String)
   return Standard.String;
```

An unbounded IDL string shall be mapped to the type **CORBA.String**.

Conforming implementations shall provide a **CORBA.Bounded_Strings** package with the same specification and semantics as **Ada.Strings.Bounded.Generic_Bounded_Length**.

The **CORBA.Bounded_Strings** package has a generic formal parameter "**Max**" declared as type **Positive** and establishes the maximum length of the bounded string at instantiation. A generic instantiation of the package shall be created using the bound for the IDL string as the associated parameter. The name and scope of the instantiation is left implementation defined.

For example, the IDL declaration:

typedef string Name;

maps to

```
type Name is new CORBA.String;
```

while the following declaration:

typedef string<512> Title;

may map to

```
with CORBA.Bounded_Strings;
package CORBA.Bounded_String_512 is new
    CORBA.Bounded_Strings(512);
```

at the library level, and

```
type Title is new CORBA.Bounded_String_512.Bounded_String;
```

in the corresponding interface package.

1.17 Mapping for Wide String Types

The IDL bounded and unbounded wide strings types are mapped to Ada's predefined wide string packages or functional equivalent.

An unbounded IDL wide string shall be mapped a derivative of the type **`CORBA.Wide_String`** or functional equivalent.

Conforming implementations shall provide a **`CORBA.Bounded_Wide_Strings`** package with the same specification and semantics as **`Ada.Strings.Wide_Bounded.Generic_Bounded_Length`**.

The **`CORBA.Bounded_Wide_Strings`** package has a generic formal parameter "**`Max`**" declared as type **`Positive`** and establishes the maximum length of the bounded string at instantiation. A generic instantiation of the package shall be created using the bound for the IDL string as the associated parameter. The name and scope of the instantiation is left implementation defined.

For example, the IDL declaration:

typedef wstring WName;

maps to

```
type WName is new CORBA.Wide_String;
```

while the following declaration:

typedef wstring<512> WTitle;

may map to

```
with CORBA.Bounded_Wide_Strings;
package CORBA.Bounded_Wide_String_512 is new
    CORBA.Bounded_Wide_Strings(512);
```

at the library level, and

```
type WTitle is new
CORBA.Bounded_Wide_String_512.Bounded_String;
```

in the corresponding interface package.

1.18 Mapping for Arrays

IDL defines multidimensional, fixed-size arrays by specifying a complex_declarator as

- any of the declarators in a typedef,
- any of the declarators in a member of a struct, or
- the declarator in any element of a union.

A complex_declarator is formed by appending one or more array size bounds to identifiers.

An IDL complex_declarator maps to an Ada array type definition. A type definition shall define an array type. The array type name shall be mapped from the identifier contained in the array_declarator prepended to "_Array." The type definition shall be an array, over the range(s) from 0 to one less then the fixed_array_size(s) specified in the array declarator, of the type mapped from the IDL type contained in the type specification. If multiple bounds are declared, a multiple dimensional array shall be created that preserves the indexing order specified in the IDL declaration. In the component definition, the name shall be mapped from the identifier contained in the array_declarator and the type shall be the array type.

SeeSection 1.13, "Mapping for Structure Types," on page 1-24, "Mapping for Union Types" on page 1-24, and Section 1.19, "Mapping for Constants," on page 1-29 for more information.

1.19 Mapping for Constants

An IDL constant shall map directly to an Ada constant. The Ada constant name shall be mapped from the identifier in the IDL declaration. The type of the Ada constant shall be mapped from the IDL const_type as specified elsewhere in this section. The value of the Ada constant shall be mapped from the IDL constant expression as specified in Section 1.3.3, "Mapping of Constant Expressions," on page 1-10. This mapping may yield a semantically equivalent literal of the correct type or a syntactically equivalent Ada expression that evaluates to the correct type and value.

For example, the following IDL constants:

```
const double Pi = 3.1415926535;
const short Line_Buffer_Length = 80;
```

shall map to

```
Pi : constant CORBA.Double := 3.1415926535;
Line_Buffer_Length : constant CORBA.Short := 80;
```

The following IDL constants:

```
const long Page_Buffer_Length =
  (Line_Buffer_Length * 60) + 2;
const long Legal_Page_Buffer_Length = (80 * 80) + 2;
```

may be mapped as

```
Page_Buffer_Length : constant CORBA.Long := 4802;
Legal_Page_Buffer_Length : constant CORBA.Long := 6402;
```

or

```
Page_Buffer_Length : constant CORBA.Long :=
                  (Line_Buffer_Length * 60) + 2;
Legal_Page_Buffer_Length : constant CORBA.Long :=
    (80 * 80) + 2;
```

1.20 Mapping for Typedefs

IDL typedefs introduce new names for types. An IDL typedef is formed from the keyword "**typedef**," a type specification, and one or more declarators. A declarator may be a simple declarator consisting of an identifier, or an array declarator consisting of an identifier and one or more fixed array sizes. An IDL typedef maps to an Ada derived type.

Each array_declarator in a typedef shall be mapped to an array type. The array type name shall be the identifier contained in the array_declarator. The type definition shall be an array over the range(s) from 0 to one less than the fixed_array_size(s) specified in the array declarator of the type mapped from the IDL type contained in the type specification. If multiple bounds are declared, a multiple dimensional array shall be created that preserves the indexing order specified in the IDL declaration.

Each simple declarator for a non-reference type (i.e., a type not in **CORBA.Object.Ref'CLASS**) shall be mapped to a derived type declaration. Each simple declarator for a reference type shall be mapped to a subtype declaration. The type name shall be the identifier provided in the simple declarator. The type definition shall be the mapping of the typespec, as specified elsewhere in this section.

For example, the following IDL typedefs:

```
typedef string Name, Street_Address[2];
typedef Name Employee_Name;
typedef enum Color {Red, Green, Blue} RGB;
interface Base {};
typedef Base Root;
```

will be mapped to

```
type Name is new CORBA.String;
type Street_Address is array(0 .. 1) of CORBA.String;
type Employee_Name is new Name;
type Color is (Red, Green, Blue);
type RGB is new Color;

subtype Root is Base.Ref;
```

1.21 Mapping for TypeCodes

TypeCodes are values that represent invocation argument types, attribute types, and Object types. They can be obtained from the Interface Repository or from IDL compilers and they have a number of uses:

- In the Dynamic Invocation interface: to indicate types of the actual arguments.
- By an Interface Repository: to represent type specifications that are part of the IDL declarations.
- As a crucial part of the semantics of the any type. Abstractly, TypeCodes consist of a “kind” field and a “parameter list.”

The Ada mapping of TypeCode is provided by the pseudo-object **`CORBA.TypeCode.Object`** type declared in the **`CORBA.TypeCode`** package nested within the `CORBA` package (see Section 1.33, “TypeCode,” on page 1-43). Its implementation is left unspecified. The primitive operations of TypeCode are mapped from the pseudo-IDL contained in the CORBA specification. These operations allow the matching of two TypeCodes, and extraction of the “kind” and “parameter list” from it. The contents of the parameter list shall be as specified in the CORBA specification.

Note – These operations do not include the ability to construct a TypeCode. Two TypeCodes are equal if the IDL type specifications from which they are compiled denote equal types. One consequence of this is that all types derived from an IDL type have equal TypeCodes.

All occurrences of type TypeCode in IDL shall be mapped to the **`CORBA.TypeCode.Object`** type.

All conforming implementations shall be capable (if asked) of generating constants of type **`CORBA.TypeCode.Object`** for all pre-defined and IDL-defined types. The name of the constant shall be “TC_” prepended to the mapped type name.

1.22 Mapping for Any Type

An Ada mapping for the IDL type `any` must fulfill two different requirements:

1. Handling values whose types are known.
2. Handling values whose types are not known at implementation compile time.

The first item covers most normal usage of the **any** type, the conversion of typed values into and out of an **any**. The second item covers situations such as those involving the reception of a request or response containing an any that holds data of a type unknown to the receiver when it was created with an Ada compiler.

The following specifies a set of Ada facilities that allows both of these cases to be handled in a type safe manner.

1.22.1 Handling Known Types

For each distinct type *T* in an IDL specification, pre-defined or IDL-defined, conforming implementations shall be capable of generating functions to insert and extract values of that type to and from type Any. The form of these functions shall be:

```
function From_Any(Item : in Any) return T;

function To_Any(Item : in T) return Any;
```

An attempt to execute **From_Any** on an **Any** value that does not contain a value of type ***T*** shall result in the raising of CORBA.Bad_Typecode.

In addition, the following function shall be defined in package CORBA:

```
function Get_Type(The_Any : in Any) return TypeCode.Ref;
```

This function allows the discovery of the type of an Any.

1.22.2 Handling Unknown Types

Certain applications may receive and wish to handle objects of type Any that contain values of a type not known at compile time, and, thus, for which a matching TypeCode constant is not available. The TypeCode facility allows the decomposition of any TypeCode to a point where all components of a type are of pre-defined (and thus known) type. In order to extract the value associated with each component of this breed of Any, conforming implementations shall provide an iterator **CORBA.Iterate_Over_Any_Elements** defined as follows:

```
generic
    with procedure Process(The_Any : in     Any;
                           Continue:    out Boolean);
procedure CORBA.Iterate_Over_Any_Elements( In_Any: in Any);
```

A conforming implementation of **Iterate_Over_Any_Elements** shall iteratively call **Process** for each component of **In_Any**. The **The_Any** argument to **Process** shall contain both the TypeCode and the value(s) of the component of the **In_Any**. Each component may itself be compound and may be of previously unknown type; therefore, the type of the component **The_Any** is another **Any**. Through the recursive use of the iterator, the input **In_Any** can be decomposed to the point that all

components are of known (eventually of pre-defined) type. At that point, a type safe conversion of the form **`From_Any`** discussed above may be applied to obtain the value of the decomposed component.

No facilities are defined or required for composing **`Any`** values of previously unknown types.

1.23 Mapping for Exception Types

An IDL exception is declared by specifying an identifier and a set of members. This member data contains descriptive information, accessible in the event the exception is raised. Standard exceptions are predefined as part of IDL and can be raised by an ORB given the occurrence of the corresponding exceptional condition. Each standard exception has member data that includes a minor code (a more detailed subcategory) and a completion status. Exceptions can also be declared that are application-specific. The raising of an application-specific exception is bound to an interface operation as part of the operation declaration. This does not imply that the corresponding implementation for the operation must raise the exception; it merely announces that the declared operation *may* raise any of the listed exception(s). A programmer has access to the value of the exception identifier upon a raise.

An application-specific exception is declared with a unique identifier (relative to the scope of the declaration) and a member list that contains zero or more IDL type declarations.

1.23.1 Exception Identifier

The IDL exception declaration shall map directly to an Ada exception declaration where the name of the Ada exception is mapped from the IDL exception identifier.

For example, the following IDL exception declaration:

```
exception null_exception{};
```

will map to the following Ada exception declaration:

```
Null_Exception: exception;
```

A programmer must be able to access the value of the exception identifier when an exception is raised. A language-defined package, **`Ada.Exceptions`**, is provided by Ada. The package contains a declaration of type **`Exception_Occurrence`**. Each occurrence of an Ada exception is represented by a distinct value of type **`Exception_Occurrence`**.

An Ada exception handler may contain a **`choice_parameter_specification`**. This declares a constant object of type **`Exception_Occurrence`**. Upon the raise of an exception, this constant represents the actual exception being handled. This constant value can be used to access the fully qualified name using the function,

Exception_Name, in the package **Ada.Exceptions**. Therefore, mapping an IDL exception declaration to an Ada exception declaration provides access to the value of the exception identifier by default.

1.23.2 Exception Members

Members are additional information available in the event of a raise of the corresponding exception. Members can contain any combination of permissible IDL types.

The following declarations shall be contained in package CORBA:

```
type IDL_Exception_Members is abstract tagged null record;

procedure Get_Members(From: in
Ada.Exceptions.Exception_Occurrence;
          To: out IDL_Exception_Members)  is abstract;
```

1.23.2.1 Standard Exceptions

A set of standard run-time exceptions is defined in the IDL language specification. Each of these exceptions has the same member form. The following IDL declarations appear for standard exceptions:

```
#define ex_body {unsigned long minor; completion_status completed;}
enum completion_status {COMPLETED_YES, COMPLETED_NO,
                        COMPLETED_MAYBE};
enum exception_type {NO_EXCEPTION, USER_EXCEPTION,
                        SYSTEM_EXCEPTION};
```

The following declarations shall exist in package CORBA:

```
type completion_Status is (COMPLETED_YES, COMPLETED_NO,
                        COMPLETED_MAYBE);
type Exception_Type is (NO_EXCEPTION, USER_EXCEPTION,
                        SYSTEM_EXCEPTION);
type System_Exception_Members is new IDL_Exception_Members with
                        record
         Minor     :  CORBA.Long;
         Completed :  Completion_Status;
  end record;
procedure Get_Members(From: in Ada.Exceptions.
                        Exception_Occurrence;
                        To: out System_Exception_Members);
```

For each standard exception specified in the CORBA specification, a corresponding Ada exception and exception members type derived from **System_Exception_Members** shall be declared in package **CORBA**. However, the name **Initialization_Failure** will be used for the Initialize exception to avoid conflict with the Ada **Initialize** procedure.

For example, the IDL standard exception declaration below:

```
exception UNKNOWN ex_body;
```

maps to the following:

```
UNKNOWN: exception;
type Unknown_Members is new System_Exception_Members
    with null record;
```

The **Unknown_Members** type will be used to hold the current values associated with the raised exception. The derived **Get_Members** function may be used to access the values.

1.23.2.2 Application-Specific Exceptions

For an application-specific exception declaration, a type extended from the abstract type, **IDL_Exception_Members**, shall be declared where the type name will be the concatenation of the exception identifier with "**_Members**." Each member shall be mapped to a component of the extension. The name used for each component shall be mapped from the member name. The type of each exception member shall be mapped from the IDL member type as specified elsewhere in this document.

The mapping shall also provide a concrete function, **Get_Members**, that returns the exception members from an object of type:

```
Ada.Exceptions.Exception_Occurrence.
```

Note – The use of the strings associated with **Exception_Message** and **Exception_Information** in the language-defined package **Ada.Exceptions** may be used by the implementor to "carry" the exception members. This may effectively render these predefined subprograms useless. If so, this fact shall be documented.

For example, the following IDL exception declaration:

```
exception access_error {
  long file_access_code;
  string access_error_description;
  }
```

will map to the following:

```
Access_error : exception;

type Access_Error_Members is new CORBA.IDL_Exception_Members
with
   record
      File_Access_Code         : CORBA.Long;
      Access_Error_Description : CORBA.String;
```

```
    end record;
procedure Get_Members(From: in
Ada.Exceptions.Exception_Occurrence;
                        To : out Access_Error_Members);
```

For consistency, the **Members** type and the **Get_Members** function must be generated even if the corresponding IDL exception has zero members. For an exception declaration without members:

```
exception a_simple_exception{};
```

the mapping will be as follows:

```
A_Simple_Exception : exception;
type A_Simple_Exception_Members is new
   CORBA.IDL_Exception_Members with null record;
procedure Get_Members(From: in
Ada.Exceptions.Exception_Occurrence;
   To: out A_Simple_Exception_Members);
```

1.23.2.3 *Example Use*

The following interface definition:

```
interface stack {
   typedef long element;
   exception overflow{long upper_bound;};
   exception underflow{};
   void push (in element the_element)
      raises (overflow);
   void pop  (out element the_element)
      raises (underflow);
};
```

maps to the following in Ada:

```
package Stack is

...

   type Element is new CORBA.Long;

   Overflow     :    exception;
   type Overflow_Members is new CORBA.IDL_Exception_Members
      with record
         Upper_Bound : CORBA.Long;
      end record;
   procedure Get_Members(From: in Ada.Exceptions.
```

```
            Exception_Occurrence;
               To:       out Overflow_Members;

      Underflow    :    exception;
      type Underflow_Members is new CORBA.IDL_Exception_Members
           with null record;
      function Get_Members(From: in Ada.Exceptions.
            Exception_Occurrence;
               To:      out  Underflow_Members);
...
end stack;
```

The following usage of the stack illustrates access to members upon an exception raise:

```
with Ada.Text_IO;
with Ada.Exceptions;
with Stack;
use Ada;

procedure Use_stack is
    ...
   The_Overflow_Members : Stack.Overflow_Members;
begin

    ...

exception

   when Stack_Error: Stack.Overflow =>
      Stack.Get_Members(Stack_Error,The_Overflow_Members;
      Text_IO.Put_Line (“Exception raised is “ &
                  Exceptions.Exception_Name (Stack_Error));
      Text_IO.Put_Line (“exceeded upper bound = “ &
   CORBA.Long’image(The_Overflow_Members.Upper_Bound));

    ...

end Use_stack;
```

1.24 Mapping for Attributes (Client-Side Specific)

Read-only attributes shall be mapped to an Ada function with name formed by prepending “**Get_**” to the mapped attribute name. Read-write attributes shall be mapped to an Ada function with name formed by prepending “**Get_**” to the mapped attribute name and an Ada procedure with name formed by prepending “**Set_**” to the mapped attribute name. The **Set** procedure takes a controlling parameter of object reference type and name “**Self**,” and a parameter with name “**To**.” The type of the **To** parameter shall be mapped from the attribute type, except for an attribute of the enclosing interface type, which shall be mapped as **Ref’CLASS**. The **Get** function

takes a controlling parameter only, of object reference type and name "**Self**." The **Get** function returns the type mapped from the attribute type, except for an attribute of the enclosing interface type, which shall be mapped as **Ref'CLASS**.

Examples of mapped attributes may be found in Section 1.8.9, "Interface Mapping Examples," on page 1-17.

1.25 Mapping for Operations (Client-Side Specific)

Operations shall map to an Ada subprogram with name mapped from the operation identifier. The first argument to operation subprograms will refer to the object that the operation is being performed on. It shall be an "**in**" mode argument with the name "**Self**" and shall be of the mapped object reference type, **Ref**.

IDL interface operations with non-void result type that have only in-mode parameters shall be mapped to Ada functions returning an Ada type mapped from the operation result type. Otherwise, (non-void IDL interface operations that have out-mode parameters, or void operations) operations shall be mapped to Ada procedures. The non-void result, if any, is returned via an added argument with name "**Returns**." This result argument shall follow all other parameter arguments.

Each specified parameter in the operation declaration and the result type shall be mapped to an argument of the mapped subprogram. The argument names shall be mapped from the parameter identifier in the IDL. The argument mode shall be preserved from the IDL. The argument or return type shall be mapped from the IDL type, except in the case of an argument or return type that is of the enclosing IDL interface type. Arguments or result types of the enclosing IDL interface type shall be mapped to Ref'CLASS. This is necessary to prevent multiple controlling parameters (it removes the ambiguity as to which parameter control dispatching.)

If an operation in an IDL specification has a context specification, then an additional argument with name "**In_Context**," of **in** mode and of type **CORBA.Context.Object** (see Section 1.32, "Context," on page 1-42) shall be added after all IDL specified arguments and before the **Returns** argument, if any. The **In_Context** argument shall have a default value of **CORBA.ORB.Get_Default_Context** (see Section 1.34, "ORB," on page 1-45).

IDL **oneway** operations are mapped the same as other operations; that is, there is no indication whether an operation is **oneway** or not in the mapped Ada specification.

Note – Implementations are encouraged to add a comment to the generated specification that states that the operation is **oneway**.

The specification of exceptions for an IDL operation is not part of the generated operation.

Examples of mapped operations may be found in Section 1.8.9, "Interface Mapping Examples," on page 1-17.

1.26 Argument Passing Considerations

The existing Ada language parameter passing conventions are followed for all types. The mapping for **in**, **out**, and **inout** parameters to the Ada "**`in`**," "**`out`**," and "**`in out`**" parameter modes obviates the need for any special parameter passing rules.

1.27 Tasking Considerations

An implementation should document whether access to CORBA services is *tasking-safe*. An operation is *tasking-safe* if two tasks within an Ada program may perform that operation and the effect is always as if they were performed in sequence.

Unless otherwise noted, it should be assumed that a CORBA operation is *not* tasking-safe, given current semantics of the CORBA specification, which is non-reentrant.

For implementations which support tasking-safe operations, the implementation should further document the blocking behavior of CORBA operations. Blocking may be at the task or program level: when an Ada task calls a CORBA operation, it is preferred that only the task, and not the whole Ada program, be blocked. Refer to the POSIX Ada binding, IEEE-Std 1003.5-1992, for further discussion.

1.28 Mapping of Pseudo-Objects to Ada

CORBA pseudo-objects are not first class objects. There are no servers associated with pseudo objects, they are not registered with an ORB, and references to pseudo-objects are not necessarily valid across computational contexts.

This mapping provides a standard binding for the pseudo-objects, the pre-defined environment for CORBA. Implementation of pseudo-objects are not specified in this mapping.

1.28.1 Mapping Rules

In general, the pseudo-objects are mapped from the pseudo-IDL according to the rules specified in preceding sections of this chapter.

The types representing pseudo-objects are not derived from **`CORBA.Object.Ref`**. Ada also supports "object semantics" better than some other OOPLs. This mapping allows the types associated with pseudo-objects are to be named **`Object`** and support copy semantics in assignment. The **`Self`** parameter will be of the **`Object`** type and **`in out`** mode, except when the operation is obviously a query-only function, in which case the **`Object`** parameter is **`in`** mode.

Conforming implementations shall raise appropriate CORBA exceptions on detection of an error condition.

Other exceptions to these general mapping rules are noted in the following text.

1.28.2 Object Semantics

Conforming implementations shall implement copy semantics for assignment of pseudo-objects of an Object type (i.e., assignment of a value of a type mapped from a pseudo-object as an Object to another object shall result in a copy of all components of the original). Conforming implementations shall implement reference semantics for assignment of pseudo-object of a Ref type (i.e., assignment of a value of a type mapped from a pseudo-object as a Ref to another object shall result in a sharing of the components of the objects).

Conforming implementations shall ensure that implementations of pseudo-objects do not “leak” memory.

1.29 NamedValue

NamedValue is used only as an element of **NVList**. **NamedValue** contains an optional name, an any value, and labeling flags. Legal flag values are ARG_IN, ARG_OUT, and ARG_INOUT, in bitwise combination with IN_COPY_VALUE. The type Flags is mapped in accordance with the mapping rules. Appropriate Flag constants must be defined by the implementation. NamedValue is mapped to a record in the CORBA package in conformance with the mapping.

```
type Flags is new CORBA.Unsigned_Long;
ARG_IN: constant Flags;
ARG_OUT: constant Flags;
ARG_INOUT: constant Flags;
IN_COPY_VALUE: constant Flags;
type NamedValue is record
   Name        : Identifier;
   Argument    : Any;
   Arg_Modes   : Flags;
end record;
```

1.30 NVList

NVList is a list of **NamedValues**. The **CORBA.NVList** package provides the mapping for the NVList pseudo-object. The **Ref** type is the mapping for the reference and, unlike most pseudo-objects, is fully a CORBA reference types. New **NamedValues** may be constructed only as part of an **NVList** through one of the **add_item** functions. An additional version of Add_Item that uses a NamedValue argument is provided.

```
package CORBA.NVList is

   type Ref is new CORBA.Object.Ref with null record;

   procedure Add_Item
      (Self       : in Ref;
      Item_Name   : in Identifier;
```

```
      Item        : in Any;
      Item_Flags  : in Flags);
   procedure Add_Item
      (Self       : in Ref;
      Item        : in NamedValue);

   -- free and free_memory are unneeded

   procedure Get_Count
         (Self    : in  Ref;
         Count    : out CORBA.Long);
private
   ... implementation defined ...
end CORBA.NVList;
```

1.31 Request

Request provides the primary support for the Dynamic Invocation Interface (DII). A new request on a particular target object may be constructed using the **Create_Request** operation in the **Object** interface. Arguments and contexts may be provided to the **Create_Request** operation or may be added after construction via the **Add_Arg** operation in the **Request** interface. Requests can be transferred to a server and responses obtained synchronously through the **Invoke** operation. The **Send** operation may be used to transfer a request to a server without waiting for results. Results, output arguments, and exceptions may be obtained later with the **Get_Response** operation. The **CORBA.Request** package provides the Ada interface to the **Request** pseudo-object and is mapped in conformance with the mapping rules, except for the arguments to **Add_Arg**. The pseudo-IDL for **Add_Arg** includes five arguments (a name, a TypeCode, a void * for the actual value, an argument length, and a Flag value) that have been replaced by a single argument of type **NamedValue** in the Ada mapping.

```
package CORBA.Request is

   type Object is private;

   procedure  Add_Arg
      (Self    : in out Object;
      Arg      : in     NamedValue);

   procedure Invoke
      (Self          : in out Object;
      Invoke_Flags   : in     Flags);

   procedure Delete
      (Self    : in out Object);

   procedure Send
      (Self          : in out Object;
      Invoke_Flags   : in     Flags);
```

```
   procedure Get_Response
      (Self             : in out Object;
      Response_Flags    : in     Flags);

private
   ... implementation defined ...
end CORBA.Request;
```

1.32 Context

A **Context** supplies optional context information associated with a method invocation. Package **CORBA.Context** provides the Ada interface for this capability. If an error in processing occurs, the CORBA system exception BAD_CONTEXT is returned. Conforming implementations must ensure adequate memory management of dynamically allocated components.

```
package CORBA.Context is

   type Ref is private;

   procedure Set_One_Value
      (Self       : in Ref;
      Prop_Name   : in Identifier;
      Value       : in CORBA.String);

   procedure Set_Values
      (Self       : in Ref;
      Values      : in CORBA.NVList.Ref);

   procedure Get_Values
      (Self          : in     Ref;
      Start_Scope    : in     Identifier;
      This_Object    : in     Boolean := TRUE;
      Prop_Name      : in     Identifier;
      Values         :     out CORBA.NVList.Ref);

   procedure Delete_Values
      (Self       : in Ref;
      Prop_Name   : in Identifier);

   procedure Create_Child
      (Self       : in     Ref;
      Ctx_Name    : in     Identifier;
      Child_Ctx   :    out Ref);

   -- Delete not needed

private
   ... implementation defined ...
```

```
end CORBA.Context;
```

1.33 TypeCode

A **TypeCode** represents IDL type information. It is intimately related to type **Any**. For this reason, package **TypeCode** that defines the **Object** type for TypeCode is a subpackage nested within the CORBA package. See Section 1.21, “Mapping for TypeCodes,” on page 1-31 for more information.

```
package CORBA is

   type TCKind is
      (tk_null, tk_void,
       tk_short, tk_long, tk_ushort, tk_ulong,
       tk_float, tk_double, tk_boolean, tk_char,
       tk_octet, tk_any, tk_TypeCode, tk_Principal,
       tk_objref,tk_struct, tk_union, tk_enum, tk_string,
       tk_sequence, tk_array, tk_alias, tk_except,
       tk_longlong, tk_ulonglong, tk_longdouble,
       tk_widechar, tk_wstring, tk_fixed,
       tk_value, tk_valuebox,
       tk_native, tk_abstract_interface);

   type ValueModifier is new Short;
   VTM_NONE        : constant ValueModifier := 0;
   VTM_CUSTOM      : constant ValueModifier := 1;
   VTM_ABSTRACT    : constant ValueModifier := 2;
   VTM_TRUNCATABLE : constant ValueModifier := 3;

   type Visibility is new Short;
   PRIVATE_MEMBER : constant Visibility := 0;
   PUBLIC_MEMBER  : constant Visibility := 1;

   package TypeCode is

      type Object is private;

      Bounds : exception;
      type Bounds_Members is new CORBA.IDL_Exception_Members
            with null record;

      procedure Get_Members

          (From : in Ada.Exceptions.Exception_Occurrence;
           To   : out Bounds_Members);

      BadKind : exception;
      type BadKind_Members is new
         CORBA.IDL_Exception_Members
            with null record;
```

```
procedure Get_Members
    (From : in Ada.Exceptions.Exception_Occurrence;
     To   : out BadKind_Members);

function Equal(Self : in Object; TC : in Object)
     return CORBA.Boolean;

function "="(Left, Right : in Object) return Boolean
   renames Equal;

function Equivalent(Self : in Object; TC : in Object)
     return CORBA.Boolean;

function Get_Compact_TypeCode(Self: in Object)
     return Object;

function Kind(Self : in Object) return TCKind;

function Id(Self : in Object)
      return CORBA.RepositoryId;

function Name(Self : in Object)
      return CORBA.Identifier;

function Member_Count(Self : in Object)
     return Unsigned_Long;
function Member_Name
    (Self  : in Object;
     Index : in CORBA.Unsigned_Long)
         return CORBA.Identifier;

function Member_Type
    (Self  : in Object;
     Index : in CORBA.Unsigned_Long) return Object;

function Member_Label
    (Self  : in Object;
    Index : in CORBA.Unsigned_Long) return CORBA.Any;
function Discriminator_Type(Self : in Object)
      return Object;
function Default_Index(Self : in Object)
      return CORBA.Long;

function Length(Self : in Object)
     return CORBA.Unsigned_Long;

function Content_Type(Self : in Object) return Object;

function Fixed_Digits(Self : in Object)
     return CORBA.Unsigned_Short;
```

```
      function Fixed_Scale(Self : in Object)
           return CORBA.Short;

      function Member_Type
          (Self  : in Object;
           Index : in CORBA.Unsigned_Long) return Object;
      function Type_Modifier(Self : in Object)
           return CORBA.ValueModifier;
      function Concrete_Base_Type(Self : in Object)
            return Object;

   end TypeCode;
```

1.34 ORB

An `ORB` is the programmer interface to the Object Request Broker. The package **`CORBA.ORB`** provides the Ada interface to the Request Broker. Package ORB is specified as a finite state machine rather than an object. None of the mapped operations contain the **`Self`** parameter specified in the pseudo-object mapping rules.

```
package CORBA is -- additional items

   type PolicyErrorCode is new Short;

   InvalidName : exception;
   type InvalidName_Members is new
      CORBA.IDL_Exception_Members
        with null record;
   procedure Get_Members
      (From : in Ada.Exceptions.Exception_Occurrence;
       To)  :    out InvalidName_Members);

   InconsistentTypeCode : exception;
   type InconsistentTypeCode_Members is
        new CORBA.IDL_Exception_Members with null record;
   procedure Get_Members
      (From : in Ada.Exceptions.Exception_Occurrence;
       To)  :    out InconsistentTypeCode_Members);

   PolicyError : exception;
   type PolicyError_Members is new
      CORBA.IDL_Exception_Members
         with null record;
   procedure Get_Members
      (From : in Ada.Exceptions.Exception_Occurrence;
       To)  :    out PolicyError_Members);
```

```
   type RepositoryId is new String;
   type Identifier   is new String;
   type ServiceType  is new Unsigned_Short;
   type ServiceOption  is new Unsigned_Long;
   type ServiceDetailType is new Unsigned_Long;

         Security : constant ServiceType := 1;

    -- ...
end CORBA;

package CORBA.ORB is

   type Octet_Sequence is new
      CORBA.Sequences.Unbounded(Octet);
   type ServiceDetail is record
        service_detail_type : ServiceDetailType;
        service_detail      : Octet_Sequence.Sequence;
   end record;

   package IDL_SEQUENCE_ServiceOption is
         new CORBA.Sequences.Unbounded(ServiceOption);
   package IDL_SEQUENCE_Service_Detail is
         new CORBA.Sequences.Unbounded(ServiceDetail);
   type ServiceInformation is record
         service_options  :
            IDL_SEQUENCE_ServiceOption.Sequence;
         service_details   :
            IDL_SEQUENCE_Service_Detail.Sequence;
   end record;
   type ObjectId is new CORBA.String;

   package IDL_SEQUENCE_ObjectId is new
        CORBA.Sequences.Unbounded(ObjectId);
   type ObjectIdList is new IDL_SEQUENCE_ObjectId.Sequence;

   function Object_To_String
      (Obj  : in CORBA.Object.Ref'CLASS)
       return CORBA.String;

   procedure String_to_Object
      (From : in     CORBA.String)
       To   : in out CORBA.Object.Ref'CLASS);

   -- Dynamic Invocation related operations

   procedure Create_List
      (Count     : in     CORBA.Long;
       New_List  :    out CORBA.NVList.Object);

   procedure Create_Operation_List
```

```
   (Oper      : in     CORBA.OperationDef.Ref;
    New_List  :    out CORBA.NVList.Object);

function Get_Default_Context
   return CORBA.Context.Object;

-- Service information operations

procedure Get_Service_Information
     (Service_Type        : in ServiceType;
      Service_Information : out ServiceInformation;
      Returns              : out Boolean);

-- initial reference operations

function List_Initial_Services return ObjectIdList;

function Resolve_Initial_References
   (Identifier : ObjectId)
     return CORBA.Object.Ref

-- TypeCode creation operations
 function create_struct_tc
     (Id      : in RepositoryId;
      Name    : in Identifier;
      Members : in StructMemberSeq)
      return CORBA.TypeCode.Object;

function create_union_tc
     (Id                 : in RepositoryId;
      Name               : in Identifier;
      Discriminator_Type : in CORBA.TypeCode.Object;
      Members            : in UnionMemberSeq)
      return CORBA.TypeCode.Object;

 function create_enum_tc
     (Id      : in RepositoryId;
      Name    : in Identifier;
      Members : in EnumMemberSeq)
      return CORBA.TypeCode.Object;

 function create_alias_tc
     (Id            : in RepositoryId;
      Name          : in Identifier;
      Original_Type : in CORBA.TypeCode.Object)
      return CORBA.TypeCode.Object;
```

```
function create_exception_tc
    (Id      : in RepositoryId;
     Name    : in Identifier;
     Members : in StructMemberSeq)
     return CORBA.TypeCode.Object;

function create_interface_tc
    (Id      : in RepositoryId;
     Name    : in Identifier)
     return CORBA.TypeCode.Object;

function create_string_tc
    (Bound : in CORBA.Unsigned_Long)
     return CORBA.TypeCode.Object;

function create_wstring_tc
    (Bound : in CORBA.Unsigned_Long)
     return CORBA.TypeCode.Object;

function create_fixed_tc
    (Digits : in CORBA.Unsigned_Short;
     Scale  : in CORBA.Short)
     return CORBA.TypeCode.Object;

function create_sequence_tc
    (Bound        : in CORBA.Unsigned_Long;
     Element_Type : in CORBA.TypeCode.Object)
     return CORBA.TypeCode.Object;

function create_recursive_sequence_tc
    (Bound        : in CORBA.Unsigned_Long;
     Offset       : in CORBA.Unsigned_Long)
     return CORBA.TypeCode.Object;

function create_array_tc
    (Length       : in CORBA.Unsigned_Long;
     Element_Type : in CORBA.TypeCode.Object)
     return CORBA.TypeCode.Object;

function create_native_tc
    (Id      : in RepositoryId;
     Name    : in Identifier)
     return CORBA.TypeCode.Object;

-- Thread related operations
function Work_Pending return Boolean;
procedure Perform_Work;
procedure Shutdown(Wait_For_Completion : in Boolean);
procedure Run;
```

```
    -- policy related operations
    function Create_Policy
        (The_Type : in PolicyType;
         Val      : Any);


end CORBA.ORB;
```

1.35 Object

Object is the root of the IDL interface hierarchy. While **Object** is a normal CORBA object (not a pseudo-object), its interface is described here because it references other pseudo-objects and its implementation will necessarily be different. The package **CORBA.Object** provides the Ada interface and includes a **Ref** type that is the root for client-side interfaces. See Section 1.8, “Mapping for Interfaces (Client-Side Specific),” on page 1-14 for more information.

```
package CORBA is

   type PolicyType is new Unsigned_Long;

   type Flags is new Unsigned_Long;

package CORBA.Object is

   type Ref is new Ada.Finalization.Controlled with private;

   function Get_Interface(Self : in Ref)
        return Ref'CLASS; -- will return
            CORBA.InterfaceDef.Ref;

   function Is_Nil(Self : in Ref) return Boolean;
   function Is_Null(Self : in Ref) return Boolean
         renames Is_Nil;

   -- duplicate unneeded, use assignment

   procedure Release(Self : in out Ref);

   function Is_A
        (Self           : Ref;
         Logical_Type_Id : String) return Boolean;

   function Non_Existent(Self : Ref) return Boolean;

   function Is_Equivalent
         (Self          : Ref;
         Other_Object   : Ref'CLASS) return Boolean;
```

```
   function Hash
      (Self    : Ref;
      Maximum : Unsigned_Long) return Unsigned_Long;

   procedure Create_Request
      (Self       : in     Ref;
      Ctx         : in     CORBA.Context.Object;
      Operation   : in     Identifier;
      Arg_list    : in     CORBA.NVList.Object;
      Result      : access NamedValue;
      Request     : out CORBA.Request.Object;
      Req_Flags   : in     Flags);

   function Get_Policy(Policy_Type : PolicyType)
        return CORBA.Policy.Ref;

   function Get_Domain_Managers(Self : Ref)
        return CORBA.DomainManager.DomainManagersList;

   type SetOverrideType is (SET_OVERRIDE, ADD_OVERRIDE);
   function Set_Policy_Overrides
      (Self       : in Ref;
      Policies    : CORBA.Policy.PolicyList;
      Set_Add     : SetOverrideType);

private
     ...
end CORBA.Object;
```

1.36 Current

Provides standardized access to computation context information. Little needed in Ada since language provides direct access to tasking and task-related information. Current references are locality constrained.

```
package CORBA.Current is

    type Ref is new CORBA.Object.Ref with null record;

private
    ... implementation defined ...
end CORBA.Current;
```

1.37 Policy

Provides access to choices that may affect operations. The Policy interface is the abstract base type for access to the various policies assigned. For example, the Security Service defines a Security Policy that is derived from this reference type.

```
package CORBA.Policy is

   type Ref is abstract new CORBA.Object.Ref with null
record;

   function Get_Policy_Type(Self: Ref) return PolicyType;
   function Copy(Self: Ref) return Ref;
   -- Destroy unneeded


   package IDL_SEQUENCE_Policy is new
      CORBA.Sequences.Unbounded
        (Ref);
   type PolicyList is new IDL_SEQUENCE_Policy.Sequence;



private

   ... implementation defined ...

end CORBA.Policy;
```

1.38 DomainManager

The domain manager provides mechanisms for:

- Establishing and navigating relationships to superior and subordinate domains.
- Creating and accessing policies.

```
package CORBA.DomainManager is

   type Ref is new CORBA.Object.Ref with null record;

   function Get_Domain_Policy
      (Self        : Ref;
      Policy_Type : PolicyType)
      return CORBA.Policy.Ref;

   package IDL_SEQUENCE_DomainManager is
      new CORBA.Sequences.Unbounded(Ref);
   type DomaingManagerList is
      new IDL_SEQUENCE_DomainManager.Sequence;

end CORBA.DomainManager;
```

1.39 ConstructionPolicy

Allows callers to assign membership of a particular object references to a domain at creation time.

```
package CORBA.ConstructionPolicy is

   type Ref is new CORBA.Policy.Ref with null record;

   procedure Make_Domain_Manager
      (Self          : in Ref;
      Object_Type    : in CORBA.InterfaceDef.Ref;
      Constr_Policy  : in Boolean);

end CORBA.ConstructionPolicy;
```

1.40 Server-Side Mapping - General

This mapping refers to the portability constraints for an implementation written in Ada as the *server side* mapping. The term *server* here is not meant to restrict implementations to the situation where method invocations cross address space or machine boundaries. This section addresses any implementation of an IDL interface.

1.41 Implementing Interfaces

The implementation of an IDL interface shall be mapped to a child package, named `Impl`, of that interface's client side interface package. The specification of this package shall contain subprograms associated with the IDL interface's operations and the declaration of a record type, **`Object`**. The operation subprograms are invoked by the ORB. The object record is used to hold member data employed by the implementation of an interface.

If the interface has no parents, the type **`Object`** shall be declared as an (implementor-defined) extension of **`PortableServer.Servant_Base`**. If the interface has a single parent, the type **`Object`** shall be an extension of the **`Object`** type mapped from the parent interface. If the interface has multiple parents the type **`Object`** shall be an extension of the **`Object`** type mapped from the first-named parent interface.

While the development and maintenance of the Impl package is explicitly the responsibility of the user, the IDL translator of a conforming implementation shall be able to generate an incomplete Impl package specification. At minimum, the package specification shall contain:

- The package declaration.
- A declaration of the **`Object`** type. The declaration shall, at least, specify the proper type derivation (as described above), but may otherwise be left incomplete.
- The specification of the primitive subprograms representing the server-side mapping of the interface's attributes and operations. The mapping rules for attributes and operations are contained below.

1.42 Implementing Operations and Attributes

The parameters passed to an implementation subprogram parallel those passed to the client side stub except that the type of the **`Self`** parameter is **`access Object`**, where **`Object`** is described above, rather than the reference type declared in the stub package.

Thus, all operation and attribute implementations will be primitive on the type **`Object`**. This allows to have a different inheritance hierarchy than that reflected in IDL. It allows inherited operations to be overridden by implementations (a facility that cannot be expressed in IDL). It also allows for alternate and delegating implementations that are not reflected in IDL.

To implement these facilities, conforming implementations are required to force all calls made to the mapped operation and attribute subprograms to be dispatching calls.

1.43 Server-Side Mapping Examples

The following IDL interface:

File cultivation.idl:

```
#include “barn.idl”

interface Plow {
   long row();
   void attach(in short blade);
   void harness(in Horse power);
};
```

causes the IDL translator to generate, in addition to the client packages discussed in previous sections, the following implementation specification:

```
with CORBA;
with CORBA.Object;
with Horse;

package Plow.Impl is
   type Object is new PortableServer.Servant_Base with
      private;
   function Row
      (Self : access Object)
      return CORBA.Long;
   procedure Attach
      (Self  : access Object;
      Blade : in CORBA.Short);
   procedure Harness
      (Self  : access Object;
      Power : in Farm.Horse.Ref);
private
```

```
   type Object is new PortableServer.Servant_Base with
         record
            -- (implementation data)
         end record;
end Plow.Impl;
```

The placement of the object record in the private part is not mandated by this mapping.

1.44 PortableServer

The subsystem root for the Portable Object Adapter.

```
package PortableServer is

   package POA_Forward is new CORBA.Forward;

   type Servant is -- ...

   function "="(Left, Right: in Servant) return Boolean;

   function Get_Default_POA (For_Servant : in Servant)
      return POA_Forward.Ref;

   package IDL_SEQUENCE_Octet is
      new CORBA.Sequences.Unbounded(CORBA.Octet);
   type ObjectId is new IDL_SEQUENCE_Octet.Sequence;

   ForwardRequest : exception;
   type ForwardRequest_Members is
      new CORBA.IDL_Exception_Members with
         record
            forward_reference : CORBA.Object.Ref;
         end record;
   procedure Get_Members
      (From    : in     CORBA.Exception_Occurrence;
      To       :    out ForwardRequest_Members);

   type ThreadPolicyValue is (ORB_CTRL_MODEL,
                              SINGLE_THREAD_MODEL);

   type LifespanPolicyValue is (TRANSIENT, PERSISTENT);

   type IdUniquenessPolicyValue is (UNIQUE_ID,MULTIPLE_ID );

   type IdAssignmentPolicyValue is (USER_ID, SYSTEM_ID);

   type ImplicitActivationPolicyValue is
         (IMPLICIT_ACTIVATION,
          NO_IMPLICIT_ACTIVATION);
```

```
   type ServantRetentionPolicyValue is (RETAIN, NON_RETAIN);

   type RequestProcessingPolicyValue is
      (USE_ACTIVE_OBJECT_MAP_ONLY, USE_DEFAULT_SERVANT,
       USE_SERVANT_MANAGER);

end PortableServer;
```

1.45 PortableServer.AdapterActivator

```
package PortableServer.AdapterActivator is

   type Ref is new CORBA.Object.Ref with  null record;

   function unknown_adapter
               (Self    : Ref;
               parent   : PortableServer.POA_Forward.Ref;
               name     : CORBA.String)
               return Boolean;

end PortableServer.AdapterActivator;
```

1.46 PortableServer.Current

```
package PortableServer.Current is

   type Ref is new CORBA.Current.Ref with null record;

   NoContext : exception;
   type NoContext_Members is new CORBA.IDL_Exception_Members
        with null record;
   procedure Get_Members
         (From : in     CORBA.Exception_Occurrence;
          To   :    out NoContext_Members);

   function get_POA(Self : Ref)
      return PortableServer.POA_Forward.Ref;

   function get_object_id(Self : Ref) return ObjectId;

end PortableServer.Current;
```

1.47 PortableServer.IdAssignmentPolicy

```
package PortableServer.IdAssignmentPolicy is

   type Ref is new CORBA.Policy.Ref with null record;
```

```
   function Get_value (Self : Ref)
      return IdAssignmentPolicyValue;

end PortableServer.IdAssignmentPolicy;
```

1.48 PortableServer.IdUniquenessPolicy

```
package PortableServer.IdUniquenessPolicy is

   type Ref is new CORBA.Policy.Ref with null record;

   function Get_value (Self : Ref)
      return IdUniquenessPolicyValue;

end PortableServer.IdUniquenessPolicy;
```

1.49 PortableServer.ImplicitActivationPolicy

```
package PortableServer.ImplicitActivationPolicy is

   type Ref is new CORBA.Policy.Ref with null record;

   function Get_value (Self : Ref)
      return ImplicitActivationPolicyValue;

end PortableServer.ImplicitActivationPolicy;
```

1.50 PortableServer.LifeSpanPolicy

```
package PortableServer.LifespanPolicy is

   type Ref is new CORBA.Policy.Ref with null record;

   function Get_value (Self : Ref)
      return LifespanPolicyValue;

end PortableServer.LifespanPolicy;
```

1.51 PortableServer.POA

```
package PortableServer.POA is

   type Ref is new CORBA.Object.Ref with null record;

   AdapterAlreadyExists : exception;
   type AdapterAlreadyExists_Members is
      new CORBA.IDL_Exception_Members with null record;
```

```
procedure Get_Members
   (From : in     CORBA.Exception_Occurrence;
   To   :    out AdapterAlreadyExists_Members);

AdapterInactive : exception;
type AdapterInactive_Members is new
   CORBA.IDL_Exception_Members
     with null record;
procedure Get_Members
   (From : in     CORBA.Exception_Occurrence;
    To   :    out AdapterInactive_Members);

AdapterNonExistent : exception;
type AdapterNonExistent_Members is
   new CORBA.IDL_Exception_Members with null record;
procedure Get_Members
   (From : in     CORBA.Exception_Occurrence;
    To   :    out AdapterNonExistent_Members);

InvalidPolicy : exception;
type InvalidPolicy_Members is
   new CORBA.IDL_Exception_Members
      with
         record
            Index : CORBA.Unsigned_Short;
      end record;
procedure Get_Members
   (From : in     CORBA.Exception_Occurrence;
    To   :    out InvalidPolicy_Members);

NoServant : exception;
type NoServant_Members is new CORBA.IDL_Exception_Members
   with null record;
procedure Get_Members
   (From : in     CORBA.Exception_Occurrence;
    To   :    out NoServant_Members);

ObjectAlreadyActive : exception;
type ObjectAlreadyActive_Members is
   new CORBA.IDL_Exception_Members with null record;
procedure Get_Members
   (From : in     CORBA.Exception_Occurrence;
    To   :    out ObjectAlreadyActive_Members);

ObjectNotActive : exception;
type ObjectNotActive_Members is
   new CORBA.IDL_Exception_Members with null record;
procedure Get_Members
   (From : in     CORBA.Exception_Occurrence;
    To   :    out ObjectNotActive_Members);
```

```
ServantAlreadyActive : exception;
type ServantAlreadyActive_Members is
   new CORBA.IDL_Exception_Members with null record;
procedure Get_Members
   (From : in     CORBA.Exception_Occurrence;
    To   :    out ServantAlreadyActive_Members);

ServantNotActive : exception;
type ServantNotActive_Members is
   new CORBA.IDL_Exception_Members with null record;
procedure Get_Members
   (From : in     CORBA.Exception_Occurrence;
    To   :    out ServantNotActive_Members);

WrongAdapter : exception;
type WrongAdapter_Members is
   new CORBA.IDL_Exception_Members with null record;
procedure Get_Members
   (From : in     CORBA.Exception_Occurrence;
    To   :    out WrongAdapter_Members);

WrongPolicy : exception;
type WrongPolicy_Members is
   new CORBA.IDL_Exception_Members with null record;
procedure Get_Members
   (From : in     CORBA.Exception_Occurrence;
    To   :    out WrongPolicy_Members);

function create_POA
   (Self             : Ref;
   adapter_name    : CORBA.String;
   a_POAManager    : PortableServer.POAManager.Ref;
   policies        : CORBA.Policy.PolicyList)
   return Ref'CLASS;

function find_POA
   (Self          : Ref;
   adapter_name : CORBA.String;
   activate_it  : CORBA.Boolean)
   return Ref'CLASS;

procedure destroy          (Self    : in Ref;
            etherealize_objects     : in CORBA.Boolean;
            wait_for_completion     : in CORBA.Boolean);

function create_thread_policy
   (Self       : Ref;
   value       : ThreadPolicyValue)
   return PortableServer.ThreadPolicy.Ref;

function create_lifespan_policy
```

```
   (Self    : Ref;
   value    : LifespanPolicyValue)
   return PortableServer.LifespanPolicy.Ref;

function create_id_uniqueness_policy
   (Self  : Ref;
   value  : IdUniquenessPolicyValue)
   return PortableServer.IdUniquenessPolicy.Ref;

function create_id_assignment_policy
   (Self    : Ref;
   value    : IdAssignmentPolicyValue)
   return PortableServer.IdAssignmentPolicy.Ref;

function create_implicit_activation_policy
   (Self    : Ref;
   value    : ImplicitActivationPolicyValue)
   return PortableServer.ImplicitActivationPolicy.Ref;

function create_servant_retention_policy
   (Self    : Ref;
   value    : ServantRetentionPolicyValue)
   return PortableServer.ServantRetentionPolicy.Ref;

function create_request_processing_policy
   (Self    : Ref;
   value    : RequestProcessingPolicyValue)
   return PortableServer.RequestProcessingPolicy.Ref;

function Get_the_name (Self : Ref) return CORBA.String;

function Get_the_parent (Self : Ref) return Ref'CLASS;

function Get_the_POAManager
   (Self : Ref)
   return PortableServer.POAManager.Ref;

function Get_the_activator
   (Self : Ref)
   return PortableServer.AdapterActivator.Ref;
procedure Set_the_activator
   (Self : in Ref;
    To   : in PortableServer.AdapterActivator.Ref);

function get_servant_manager (Self : Ref)
   return PortableServer.ServantManager.Ref;

procedure set_servant_manager
   (Self : in     Ref;
   imgr  : in     PortableServer.ServantManager.Ref);
```

```
function get_servant (Self : Ref) return Servant;

procedure set_servant
   (Self : in Ref; p_servant : in Servant);

function activate_object
   (Self       : Ref;
   p_servant   : Servant)
   return ObjectId;

procedure activate_object_with_id
   (Self       : in     Ref;
   id          : in     ObjectId;
   p_servant   : in     Servant);

procedure deactivate_object
   (Self    : in     Ref;
   oid      : in     ObjectId);

function create_reference
   (Self    : Ref;
   intf     : CORBA.RepositoryId)
   return CORBA.Object.Ref;

function create_reference_with_id
   (Self    : Ref;
   oid      : ObjectId;
   intf     : CORBA.RepositoryId)
   return CORBA.Object.Ref;

function servant_to_id
   (Self       : Ref;
   p_servant   : Servant)
   return ObjectId;

function servant_to_reference
   (Self       : Ref;
   p_servant   : Servant)
   return CORBA.Object.Ref;

function reference_to_servant
   (Self       : Ref;
   reference   : CORBA.Object.Ref'CLASS)
   return Servant;

function reference_to_id
   (Self       : Ref;
   reference   : CORBA.Object.Ref'CLASS)
   return ObjectId;

function id_to_servant
```

```
      (Self : Ref;
      oid   : ObjectId)
      return Servant;

   function id_to_reference
      (Self : Ref;
      oid  : ObjectId)
      return CORBA.Object.Ref;

   package Convert is new
         PortableServer.POA_Forward.Convert(Ref);

end PortableServer.POA;
```

1.52 PortableServer.POAManager

```
package PortableServer.POAManager is

   type Ref is new CORBA.Object.Ref with null record;

   AdapterInactive : exception;
   type AdapterInactive_Members is
      new CORBA.IDL_Exception_Members with null record;
   procedure Get_Members
      (From : in     CORBA.Exception_Occurrence;
       To   :    out AdapterInactive_Members);

   procedure activate (Self : in     Ref);

   procedure hold_requests
      (Self                  : in     Ref;
      wait_for_completion    : in     CORBA.Boolean);

   procedure discard_requests
      (Self                  : in     Ref;
      wait_for_completion    : in     CORBA.Boolean);

   procedure deactivate
      (Self                  : in     Ref;
      etherealize_objects    : in     CORBA.Boolean;
      wait_for_completion    : in     CORBA.Boolean);

end PortableServer.POAManager;
```

1.53 PortableServer.RequestProcessingPolicy

```
package PortableServer.RequestProcessingPolicy is

   type Ref is new CORBA.Policy.Ref with null record;
```

```
   function Get_value (Self : Ref)
      return RequestProcessingPolicyValue;

end PortableServer.RequestProcessingPolicy;
```

1.54 PortableServer.ServantActivator

```
package PortableServer.ServantActivator is

   type Ref is new PortableServer.ServantManager.Ref
      with null record;

   function incarnate
      (Self       : in Ref;
      oid         : in ObjectId;
      adapter     : in PortableServer.POA.Ref)
      return Servant;

   procedure etheralize
      (Self       : in Ref;
      oid         : in PortableServer.ObjectId;
      adapter     : in PortableServer.POA_Forward.Ref;
      serv        : in PortableServer.Servant;
      cleanup_in_progress     : in CORBA.Boolean;
      remaining_activations   : in CORBA.Boolean);

end PortableServer.ServantActivator;
```

1.55 PortableServer.ServantLocator

```
package PortableServer.ServantLocator is

   type Ref is new PortableServer.ServantManager.Ref
      with null record;

   type Cookie is -- ... native

   procedure preinvoke
      (Self       : in     Ref;
      oid         : in     ObjectId;
      adapter     : in     PortableServer.POA.Ref;
      operation   : in     CORBA.Identifier;
      the_cookie  :    out Cookie;
      Returns     :    out Servant);

   procedure postinvoke
      (Self        : in     Ref;
       oid         : in     ObjectId;
```

```
      adapter     : in     PortableServer.POA.Ref;
      operation   : in     CORBA.Identifier;
      the_cookie  : in     Cookie;
      the_servant : in     Servant);

end PortableServer.ServantLocator;
```

1.56 PortableServer.ServantManager

```
package PortableServer.ServantManager is

   type Ref is new CORBA.Object.Ref with null record;

end PortableServer.ServantManager;
```

1.57 PortableServer.ServantRetentionPolicy

```
package PortableServer.ServantRetentionPolicy is

   type Ref is new CORBA.Policy.Ref with null record;

   function Get_value (Self : Ref)
      return ServantRetentionPolicyValue;

end PortableServer.ServantRetentionPolicy;
```

1.58 PortableServer.ThreadPolicy

```
package PortableServer.ThreadPolicy is

   type Ref is new CORBA.Policy.Ref with null record;

   function Get_value (Self : Ref) return ThreadPolicyValue;

end PortableServer.ThreadPolicy;
```

Appendix A Glossary of Ada Terms

A.1 Glossary Terms

This appendix defines terms used in the document that are not defined in the glossary of the CORBA specification. These definitions are quoted mostly from the Ada 95 Reference Manual (ISO/IEC 8652:1995).

Class

A class is a set of types that is closed under derivation, which means that if a given type is in the class, then all types derived from that type are also in the class. The set of types of a class share common properties, such as their primitive operations.

Class-wide types

Class-wide types are defined for (and belong to) each derivation class rooted at a tagged type. Given a subtype S of a tagged type T, S'Class is the subtype_mark for a corresponding subtype of the tagged class-wide type T'Class. Such types are called "class-wide" because when a formal parameter is defined to be of a class-wide type T'Class, an actual parameter of any type in the derivation class rooted at T is acceptable.

Controlled type

A controlled type supports user-defined assignment and finalization. Objects are always finalized before being destroyed.

Package

Packages are program units that allow the specification of groups of logically related entities. Typically, a package contains the declaration of a type along with the declarations of primitive subprograms of the type, which can be called from outside the package, while the inner working remains hidden from outside users.

Primitive operations

The primitive operations of a type are the operations (such as subprograms) declared together with the type declaration. They are inherited by other types in the same class of types. For a tagged type, the primitive subprograms are dispatching subprograms, providing run-time polymorphism. A dispatching subprogram may be called with statically tagged operands, in which case the subprogram body invoked is determined at compile time. Alternatively, a dispatching subprogram may be called using a dispatching call, in which case the subprogram body invoked is determined at run time.

Subsystems

A library unit is a "top-level" separately compiled program unit, and is always a package, subprogram, or generic unit. Library units may have other (logically nested) library units as children, and may have other program units physically nested within them. A root library unit, together with its children and grandchildren and so on, form a subsystem.

Tagged type

The values of a tagged type have a run-time type tag, which indicates the specific type from which the value originated. An operand of a class-wide tagged type can be used in a dispatching call; the tag indicates which subprogram body to invoke.

Withing, withs, with clause

The Ada mechanism to gain visibility to a compilation unit is to include a “with clause” naming that compilation unit. Such a compilation unit is said to be “withed” by the current unit. Conversely, the current unit “withs” the named unit. This “withing” allows use of declarations from the “withed” unit through a “selected component” notation consisting of the withed unit name, “.”, and the declaration name.

Index