

Contents

This chapter contains the following sections.

Section Title	Page
“Overview”	23-1
“Mapping Summary”	23-2
“Other Mapping Requirements”	23-5
“Lexical Mapping”	23-6
“Mapping of IDL to Ada”	23-10
“Mapping of Pseudo-Objects to Ada”	23-36
“Server-Side Mapping”	23-43
“Predefined Language Environment: Subsystem CORBA”	23-45
“Glossary of Ada Terms”	23-65

23.1 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.

23.1.1 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.

23.2 Mapping Summary

Table 23-1 summarizes the mapping of IDL constructs to Ada constructs. The following sections elaborate on each of these constructs.

Table 23-1 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

23.2.1 Interfaces and Tagged Types

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.

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.

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.

23.2.2 *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.

23.2.3 *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.

23.2.4 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.

23.2.5 Data Types

The mapping of types is summarized in Table 23-2.

Table 23-2 Summary of Mapping Types

Type(s)	Mapping
Arithmetic	Corresponding Ada arithmetic types
char	Character
boolean	Boolean
octet	Interfaces.Unsigned_8
any	CORBA.Any (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

23.2.6 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.

23.2.7 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.

23.3 Other Mapping Requirements

23.3.1 Implementation Considerations

The Ada language mapping can be implemented in a number of ways. Stub packages, ORB packages, and datatype packages 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.

23.3.2 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.

23.3.3 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.

23.3.4 Tasking

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

23.4 *Lexical Mapping*

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

23.4.1 *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:

- 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.

23.4.2 *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.

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 “#.”

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.

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 23-3 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 23-3 Lexical Equivalents for the Defined Escape Sequences

Description	IDL Escape Sequence	ISO 646 Octal Value	Ada Lexical Mapping
newline	\n	012	Ada.Characters.Latin_1.LF
horizontal tab	\t	011	Ada.Characters.Latin_1.HT
vertical tab	\v	013	Ada.Characters.Latin_1.VT
backspace	\b	010	Ada.Characters.Latin_1.BS
carriage return	\r	015	Ada.Characters.Latin_1.CR
form feed	\f	014	Ada.Characters.Latin_1.FF
alert	\a	007	Ada.Characters.Latin_1.BEL
backslash	\\	134	Ada.Characters.Latin_1.Reverse_Solidus
question mark	\?	077	Ada.Characters.Latin_1.Question
single quote	\'	047	Ada.Characters.Latin_1.Apostrophe

Table 23-3 Lexical Equivalents for the Defined Escape Sequences

Description	IDL Escape Sequence	ISO 646 Octal Value	Ada Lexical Mapping
double quote	\"	042	Ada.Characters.Latin_1.Quotation
octal number	\ooo	ooo	Character'val(8#ooo#)
hex number	\xhh	Octal equivalent to the hexadecimal number hh	Character'val(16#hh#)

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.

23.4.3 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.

Mapping of Operators

Table 23-4 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 23-4 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.

An integer constant expression is evaluated as unsigned long unless it contains a negated integer literal or the name of an integer constant with a negative value. In the latter case, the constant expression is evaluated as

signed long. The computed value is coerced back to the target type in constant initializers. It is an error if the computed value exceeds the range of the evaluated-as type (long or unsigned long).

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

23.5 Mapping of IDL to Ada

This section specifies the syntactic and semantic mapping of OMG IDL to Ada. Unless noted, the mapping is applicable to both client-side and server-side interfaces. Mapping considerations unique to the server-side interface are specified in “Server-Side Mapping” on page 23-43.

23.5.1 Names

Identifiers

The lexical mapping of IDL identifiers is specified in “Mapping of Identifiers” on page 23-6. All identifiers in the Ada interfaces generated from IDL shall be mapped from the corresponding IDL identifiers.

Scoped Names

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

- The subsystem name, if specified, forms an Ada library package.
- 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.

23.5.2 IDL Files

Subsystems

Subsystems are expressed in Ada by hierarchies of packages and child packages. The closest corresponding construct in IDL is the “module” which defines a scope that can contain other modules, interfaces, and definitions. However, at least in Revision 1.2, a module may not extend across a file boundary. This is a serious limitation in the ability of a provider of a set of capabilities to prevent name clashes with other subsystems. For this reason, support for the generation of a subsystem is defined.

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.

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.

Other Pre-Processing

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

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 the file is part

of a subsystem, the global name scope shall be mapped as a child of the (implied) subsystem package. 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.

23.5.3 *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.

23.5.4 *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.

23.5.5 *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 “Interfaces and Inheritance” on page 23-13.

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

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).

Interfaces and Inheritance

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

Let `C` be derived from `P1 . . . Pn`, where for each `i`, `Pi` is an interface. Let `OP1(Pi) . . . OPm(Pi)` be the operations specified for `Pi`. Then `C`'s mapping will be a package which will contain an `OP(C)` for each `OPj(Pi)` where `i` is 2 to `n`. The `OPj(P1)` operations are inherited using Ada's inheritance mechanism.

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_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.

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 “Object” on page 23-42) as follows:

```

-- Duplicate unneeded, use assignment

function Is_Nil(Self : Ref) return Boolean;

procedure Release(Self : Ref'CLASS);

```

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

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);
```

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 `Constraint_Error`.

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 uninitialized object references, and does require or allow definition of a Nil constant.

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 “Object” on page 23-42 for more information.

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;
};
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 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;
    package Convert is new Egg_Forward.Convert(Ref);
    function To_Ref(The_Ref : in CORBA.Object.Ref'CLASS)
        return 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;
  package Convert is new Chicken_Forward.Convert(Ref);
  function To_Ref(The_Ref : in CORBA.Object.Ref'CLASS)
    return 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;
```

23.5.6 Mapping for Types

IDL is a typed language, but weakly typed. The following subsections specify the mapping of IDL types to corresponding Ada types.

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 arithmetic types and string.

Mapping for Arithmetic Types

Several basic arithmetic 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 23-5.

Table 23-5 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 $-(2^{31}) .. (2^{31} - 1)$
Unsigned_Short	unsigned short	integer, range $0 .. (2^{16} - 1)$
Unsigned_Long	unsigned long	integer, range $0 .. (2^{32} - 1)$
Float	float	floating point, ANSI/IEEE 754-1985 single precision
Double	double	floating point, ANSI/IEEE 754-1985 double precision
Char	char	8 bit ISO Latin-1 (8859.1) 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 23-6, should be used.

Table 23-6 Declarations

Ada Type	Definition
CORBA.Short	type Short is new Interfaces.Integer_16;
CORBA.Long	type Long is new Interfaces.Integer_32;
CORBA.Unsigned_Short	type Unsigned_Short is new Interfaces.Unsigned_16;
CORBA.Unsigned_Long	type Unsigned_Long is new Interfaces.Unsigned_32;
CORBA.Float	type Float is new Interfaces.IEEE_Float_32;
CORBA.Double	type Double is new Interfaces.IEEE_Float_64;
CORBA.Char	subtype Char is Standard.Character;
CORBA.Octet	type Octet is new Interfaces.Unsigned_8;

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

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;
```

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);
```

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;

```

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 “Mapping for Structure Types” on page 23-22 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;

```

Mapping for Sequence Types

Two template types are predefined: sequence and string. 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` (see Appendix A - “Package CORBA.Sequences” on page 23-55) 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 first or only actual argument will be the type mapped from the `simple_type_spec`. For a bounded sequence, the second formal shall be a constant mapped from the `positive_int_constant`. 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;
    type IDL_SEQUENCE_octet_Array is
        is array (Integer range <>) of CORBA.Octet;
    package IDL_SEQUENCE_octet is
        new CORBA.Sequences.Unbounded
            (CORBA.Octet, IDL_SEQUENCE_Octet_Array);
    type Data8 is new IDL_SEQUENCE_octet.Sequence;

    type IDL_SEQUENCE_long_Array is
        is array (Integer range <>) of CORBA.Long;
    package IDL_SEQUENCE_1024_long is
        new CORBA.Sequences.Bounded
            (CORBA.Long, IDL_SEQUENCE_long_Array, 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 23-28) 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.Unbounded;

```

...
type IDL_SEQUENCE_octet_Array is
    array (Integer range <>) of CORBA.Octet;
package IDL_SEQUENCE_octet is
    CORBA.Sequences.Unbounded
        (CORBA.Octet, IDL_SEQUENCE_octet_Array);

type IDL_SEQUENCE_SEQUENCE_octet_Array is
    array (Integer range <>) of IDL_SEQUENCE_octet.Sequence;
package IDL_SEQUENCE_SEQUENCE_octet is
    new CORBA.Sequences.Unbounded
        (IDL_SEQUENCE_octet.Sequence,
         IDL_SEQUENCE_SEQUENCE_octet_Array);

type Ragged8 is new IDL_SEQUENCE_SEQUENCE_octet.Sequence

```

Mapping for String Types

The IDL bounded and unbounded strings types are mapped to Ada's predefined string packages rooted at `Ada.Strings`.

An unbounded IDL string shall be mapped directly to the type `CORBA.String`. This type shall be defined as:

package CORBA is

...

**type String is new
Ada.Strings.Unbounded.Unbounded_String;**

...

end CORBA;

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.

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 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.

See “Mapping for Structure Types” on page 23-22, “Mapping for Union Types” on page 23-23, and “Mapping for Constants” on page 23-27 for more information.

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 “Mapping of Constant Expressions” on page 23-8. 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;

```

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 shall be mapped to a derived type 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 previously 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;

```

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;

```

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 “`TypeCode`” on page 23-40). 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.

23.5.7 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.

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 `Constraint_Error`.

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.

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.

23.5.8 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.

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.

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;
```

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_Exception_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.

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);
```

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;

```

23.5.9 Mapping for Operations and Attributes (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.”

If appropriate, 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 and the argument shall be of type mapped from the IDL type.

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 “Context” on page 23-39) 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 “ORB” on page 23-42).

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 the same type as the attribute and name “To.” The Get function takes a controlling parameter only (of object reference type and name “Self”) and returns the type mapped from the attribute type.

IDL oneway operations are mapped the same as other operations; that is, there is no way to know by looking at the Ada whether an operation is oneway or not.

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 and attributes may be found in “Interface Mapping Examples” on page 23-16.

23.5.10 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 removes the need for any special parameter passing rules.

23.5.11 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.

23.6 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.

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 specifies that 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.

Status result types are generally not needed by Ada. 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.

Object Semantics

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

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

23.6.1 NamedValue

NamedValue is used only as an element of NVList. NamedValue contains an optional name, an any value, and labelling 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;
    Len       : Long;
    Arg_Modes : Flags;
end record;

```

23.6.2 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. 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 Object is private;

    procedure Add_Item
        (Self      : in out Object;
         Item_Name  : in      Identifier;
         Item       : in      Any;
         Item_Flags : in      Flags);
    procedure Add_Item
        (Self      : in out Object;
         Item       : in      NamedValue);

    -- free and free_memory are unneeded

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

```

23.6.3 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;

```

23.6.4 Context

A Context supplies optional context information associated with a method invocation. Package CORBA.Context provides the Ada interface for this capability and is mapped in accordance with the mapping rules. If an error in processing occurs, the CORBA system exception BAD_CONTEXT is returned.

```

package CORBA.Context is

  type Object is limited private;

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

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

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

```

```

procedure Delete_Values
  (Self      : in out Object;
   Prop_Name : in      Identifier);

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

procedure Delete
  (Self      : in      Object;
   Delete_Desendents : in      Boolean := FALSE);

private
  ... implementation defined ...
end CORBA.Context;

```

23.6.5 *Principal*

A *Principal* represents information about principals requesting operations. There are no defined operations in the CORBA specification. Package `CORBA.Principal` provides the Ada interface and is mapped in accordance with the mapping rules. Because type `Principal` may be passed as a parameter, functions supporting conversion to type `Any` are provided.

```

package CORBA.Principal is

  type Object is private;

  function To_Any (From : in Object) return Any;
  function From_Any(From : in Any) return Object;

  function Is_Principal (Item : Any) return Boolean;

  -- implementations may add operations

end CORBA.Principal;

```

23.6.6 *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 “Mapping for TypeCodes” on page 23-28 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);

  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);

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

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

    function Kind(Self : in Object) return TCKind;

    function Param_Count(Self : in Object) return CORBA.Long;

    function Parameter
      (Self : in      Object;
       Index : in     CORBA.Long) -- note origin is 0
      return Any;

  end TypeCode;

end CORBA;

```

23.6.7 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.ORB is

    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);

    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;

end CORBA.ORB;
```

23.6.8 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 “Mapping for Interfaces (Client-Side Specific)” on page 23-12 for more information.

```

package CORBA.Object is

    type Ref is tagged private;

    function To_Any (From : in Ref) return Any;
    function From_Any(From : in Any) return Ref;

    function Get_Implementation(Self : in Ref)
        return CORBA.ImplementationDef.Ref;

    function Get_Interface(Self : in Ref)
        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 Create_Request
        (Self      : in    Ref;
         Ctx       : in    CORBA.Context.Object;
         Operation : in    Identifier;
         Arg_list  : in    CORBA.NVList.Object;
         Result    : in out NamedValue;
         Request   :      out CORBA.Request.Object;
         Req_Flags : in    Flags;
         Returns   :      out Status);

private
    ...
end CORBA.Object;

```

23.6.9 Environment

The Environment pseudo-object is not needed by this mapping except as a parameter to the Get_Principal operation in the BOA interface. The CORBA.Environment package provides an Ada interface to which implementations may add additional operations.

```

package CORBA.Environment is

    type Object is private;

private
    ... implementation defined ...
end CORBA.Environment;

```

23.7 Server-Side Mapping

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.

The current CORBA specification covers only a subset of the functionality needed to build a server. As a consequence, it is unlikely that a conforming, working server can be guaranteed to be portable. However, we expect the bulk of the server code to be portable from one ORB implementation to another.

23.7.1 *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 `CORBA::Implementation_Defined::Object` where *Implementation_Defined* is implementation dependent. If the interface has a single parent, the type `Object` shall be an extension of the `Object` type mapped from the parent interface.

23.7.2 *Implementing Operations and Attributes*

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

23.7.3 *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);  
};
```

#pragma Subsystem("Farm");

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 Farm.Horse;
package Farm.Plow.Impl is
    type Object is new CORBA.Implementation_Defined.Object 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 CORBA.Object.Object with
        record
            -- (implementation data)
        end record;
end Farm.Plow.Impl;

```

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

23.8 *Predefined Language Environment: Subsystem CORBA*

This appendix provides a complete specification of the CORBA package and its children that comprise the pre-defined Ada environment which CORBA-compliant clients and servers must be provided by compliant products.

Any references to package `Implementation_Defined` shown here indicate items that are to be defined by the implementation and should not be misinterpreted as the required definitions for these items. All types derived from `Implementation_Defined.Opaque_Type` are completely implementation defined and should be made private. Implementations are allowed to add definitions required by extensions of the CORBA specification implemented by ORB products. Other allowable additions include, but are not limited to, representation clauses and additional with clauses.

23.8.1 *Package CORBA*

```

with Ada.Exceptions;
with Ada.Strings.Unbounded;
with Implementation_Defined; -- dummy package to let compile succeed
with Interfaces;

--I module CORBA {

```

```

package CORBA is

-- CORBA Module: In order to prevent names defined with the
-- CORBA specification from clashing with names in programming languages
-- and other software systems, all names defined by CORBA are treated as
-- if they were defined with a module named CORBA.

    -- Each IDL data type is mapped to a native data
    -- type via the appropriate language mapping.
    -- The following definitions may differ. See the mapping
    -- specification for more information.
    subtype Boolean      is Standard.Boolean;
    type   Short         is new Interfaces.Integer_16;
    type   Long          is new Interfaces.Integer_32;
    type   Unsigned_Short is new Interfaces.Unsigned_16;
    type   Unsigned_Long  is new Interfaces.Unsigned_32;
    type   Float         is new Interfaces.IEEE_Float_32;
    type   Double        is new Interfaces.IEEE_Float_64;
    subtype Char         is Standard.Character;
    type   Octet         is new Interfaces.Unsigned_8;
    type   String        is new Ada.Strings.Unbounded.Unbounded_String;

    -- Exceptions

    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;

    -- Standard Exceptions:
    --I #define ex_body{ unsigned long minor, completion_status completed;}
    --I enum completion_status{COMPLETED_YES, COMPLETED_NO, COMPLETED_MAYBE};
    type Completion_Status is
        (COMPLETED_YES, COMPLETED_NO, COMPLETED_MAYBE);

    --I enum exception_type{ NO_EXCEPTION, USER_EXCEPTION, SYSTEM_EXCEPTION};
    type Exception_Type is
        (NO_EXCEPTION, USER_EXCEPTION, SYSTEM_EXCEPTION);

    type Ex_Body is new CORBA.IDL_Exception_Members with record
        Minor: CORBA.Unsigned_Long;
        Completed : Completion_Status;
    end record;

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

    --I exception UNKNOWN          ex_body; // the unknown exception
    UNKNOWN: exception;
    --I exception BAD_PARAM        ex_body; // an invalid parameter was passed
    BAD_PARAM: exception;
    --I exception NO_MEMORY        ex_body; // dynamic memory allocation failure
    NO_MEMORY: exception;
    --I exception IMP_LIMIT        ex_body; // violated implementation limit
    IMP_LIMIT: exception;
    --I exception COMM_FAILURE     ex_body; // communication failure
    COMM_FAILURE: exception;
    --I exception INV_OBJREF       ex_body; // invalid object reference

```

```

INV_OBJREF: exception;
--I exception NO_PERMISSION      ex_body; // no permission for attempted op.
NO_PERMISSION: exception;
--I exception INTERNAL          ex_body; // ORB internal error
INTERNAL: exception;
--I exception MARSHAL           ex_body; // error marshalling param/result
MARSHAL: exception;
--I exception INITIALIZE        ex_body; // ORB initialization failure
INITIALIZATION_FAILURE : exception;
--I exception NO_IMPLEMENT      ex_body;
--// operation implementation unavailable

NO_IMPLEMENT: exception;
--I exception BAD_TYPECODE      ex_body; // bad typecode
BAD_TYPECODE: exception;
--I exception BAD_OPERATION     ex_body; // invalid operation
BAD_OPERATION: exception;
--I exception NO_RESOURCES      ex_body; // insufficient resources for req.
NO_RESOURCES: exception;
--I exception NO_RESPONSE       ex_body;
--// response to request not yet available

NO_RESPONSE: exception;
--I exception PERSIST_STORE     ex_body; // persistent storage failure
PERSIST_STORE: exception;
--I exception BAD_INV_ORDER     ex_body; // routine invocations out of order
BAD_INV_ORDER: exception;
--I exception TRANSIENT         ex_body;
--// transient failure - reissue request

TRANSIENT: exception;
--I exception FREE_MEM          ex_body; // cannot free memory
FREE_MEM: exception;
--I exception INV_IDENT         ex_body; // invalid identifier syntax
INV_IDENT: exception;
--I exception INV_FLAG          ex_body; // invalid flag was specified
INV_FLAG: exception;
--I exception INTF_REPOS        ex_body;
--// error accessing interface repository

INTF_REPOS: exception;
--I exception BAD_CONTEXT       ex_body; // error processing context object
BAD_CONTEXT: exception;
--I exception OBJ_ADAPTER       ex_body; // failure detected by object adapter
OBJ_ADAPTER: exception;
--I exception DATA_CONVERSION ex_body; // data conversion error
DATA_CONVERSION: exception;

-- TypeCodes

--I enum TCKind {
--I      tk_null, tk_void,
--I      tk_short, tk_long, tk_ushort, tk_ulong,
--I      tk_float, tk_double, tk_boolean, tk_char,
--I      tk_octet, tk_any, tk_TypeCode, tk_Principal, tk_objref,
--I      tk_struct, tk_union, tk_enum, tk_string,
--I      tk_sequence, tk_array
--I };
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);

-- Any Type: The any type permits the specification of
-- values that can express any IDL type.
type Any is private;

--I interface TypeCode {
package TypeCode is

    type Object is private;

    --I exception Bounds {};
    Bounds : exception;
    type Bounds_Members is new CORBA.IDL_Exception_Members
        with null record;
    function Get_Members
        (X : Ada.Exceptions.Exception_Occurrence) return Bounds_Members;

    --I boolean equal (in TypeCode tc);
    function Equal(Self : in Object; TC : in Object) return CORBA.Boolean;
    function "="(Left, Right : in Object) return Boolean renames Equal;

    --I TCKind kind ();
    function Kind(Self : in Object) return TCKind;

    --I long param_count ();
    --I // The number of parameters for this TypeCode.
    function Param_Count(Self : in Object) return CORBA.Long;

    --I any parameter (in long index) raises (Bounds);
    --I // The index'th parameter. Parameters are indexed
    --I // from 0 to (param_count-1).
    function Parameter
        (Self : in Object;
         Index : in CORBA.Long) -- note origin is 0
        return Any;
    --I };

private
    ... implementation defined ...
end TypeCode;

function Get_Type(The_Any : in Any) return Typecode.Object;

function To_Any (From : in Octet)          return Any;
function To_Any (From : in Short)         return Any;
function To_Any (From : in Long)          return Any;

```



```

function To_Any (From : in Unsigned_Short) return Any;
function To_Any (From : in Unsigned_Long) return Any;
function To_Any (From : in Boolean) return Any;
function To_Any (From : in Char) return Any;
function To_Any (From : in String) return Any;

function From_Any (From: in Any) return Octet;
function From_Any (From: in Any) return Short;
function From_Any (From: in Any) return Long;
function From_Any (From: in Any) return Unsigned_Short;
function From_Any (From: in Any) return Unsigned_Long;
function From_Any (From: in Any) return Boolean;
function From_Any (From: in Any) return Char;
function From_Any (From: in Any) return String;

--I typedef string Identifier;
type Identifier is new CORBA.String;

-- Dynamic Invocation Interface
-- Common Data Structures

--I typedef unsigned long Flags;
type Flags is new CORBA.Unsigned_Long;
ARG_IN : constant Flags;
ARG_OUT : constant Flags;
ARG_INOUT: constant Flags;

--I struct NamedValue {
--I     Identifier name;           // argument name
--I     any argument;             // argument
--I     long len;                  // length/count of argument value
--I     Flags arg_modes;           // argument mode flags
--I };
type NamedValue is record
    Name : Identifier;
    Argument : Any;
    Len : Long;
    Arg_Modes : Flags;
end record;

OUT_LIST_MEMORY: constant Flags; -- CORBA 6.2.1
IN_COPY_VALUE: constant Flags; -- CORBA 6.2.2
INV_NO_RESPONSE: constant Flags; -- CORBA 6.3.1
INV_TERM_ON_ERR: constant Flags; -- CORBA 6.3.2
RESP_NO_WAIT: constant Flags; -- CORBA 6.3.3
DEPENDENT_LIST: constant Flags; -- CORBA 6.4.2
CTX_RESTRICT_SCOPE: constant Flags; -- CORBA 6.6.4

-- Container and Contained Objects
-- moved to child package CORBA.Repository_Root

--I typedef unsigned long Status;
type Status is new CORBA.Unsigned_Long;

private

    ... implementation defined ...
end CORBA;

```

23.8.2 Package *CORBA.Bounded_Strings*;

```
--with Ada.Strings.Bounded;
--package CORBA.Bounded_Strings
--    renames Ada.Strings.Bounded.Generic_Bounded_Length;
```

Note – Because library units must be renames of library units and because `Generic_Bounded_Length` is not a library unit, conforming implementations must provide a substitute.

23.8.3 Package *CORBA.Context*

```
--I interface Context {
with CORBA.NVList;
package CORBA.Context is

    type Object is limited private;

--I     Status set_one_value (
--I         in Identifierprop_name,    // property name to add
--I         in stringvalue             // property value to add
--I     );
    procedure Set_One_Value
        (Self      : in out Object;
         Prop_Name : in     Identifier;
         Value     : in     CORBA.String);

--I     Status set_values (
--I         in NVListvalues             // property values to be changed
--I     );
    procedure Set_Values
        (Self      : in out Object;
         Values    : in     CORBA.NVList.Object);

--I     Status get_values (
--I         in Identifierstart_scope, // search scope
--I         in Flagsop_flags,        // operation flags
--I         in Identifierprop_name,  // name of property(s) to retrieve
--I         out NVListvalues         // requested property(s)
--I     );
    procedure Get_Values
        (Self      : in out Object;
         Start_Scope : in     Identifier;
         Op_Flags   : in     Flags;
         Prop_Name  : in     Identifier;
         Values     : out     CORBA.NVList.Object);

--I     Status delete_values (
--I         in Identifierprop_name    // name of property(s) to delete
--I     );
    procedure Delete_Values
        (Self      : in out Object;
         Prop_Name : in     Identifier);

--I     Status create_child (
--I         in Identifierctx_name,    // name of context object
--I         out Contextchild_ctx     // newly created context object
```

```

--I      );
--I      procedure Create_Child
--I      (Self      : in out Object;
--I      Ctx_Name   : in      Identifier;
--I      Child_Ctx  : out Object);

--I      Status delete (
--I      in Flagsdel_flags      // flags controlling deletion
--I      );
--I      };
--I      procedure Delete
--I      (Self      : in out Object;
--I      Del_Flags : in      Flags);

private
  ... implementation defined ...
end CORBA.Context;

```

23.8.4 Package *CORBA.Environment*

```

--I interface Environment {};
package CORBA.Environment is

  type Object is private;

private
  ... implementation defined ...
end CORBA.Environment;

```

23.8.5 Package *CORBA.Forward*

```

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_Forward (The_Ref : in Ref_Type) return Ref;
  end Convert;
private
  function To_Ref(From : in Any) return Ref;
end CORBA.Forward;

```

23.8.6 Package *CORBA.Iterate_Over_Any_Elements*

```

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

```

23.8.7 Package CORBA.NVList

```
--I interface NVList {
package CORBA.NVList is

    type Object is private;

--I Status add_item (
--I     in Identifier    item_name, // name of item
--I     in TypeCode     item_type, // item datatype
--I     in void          *value,    // item value
--I     in long          value_len, // length of item value
--I     in Flags         item_flags // item flags
--I );
    procedure Add_Item
        (Self      : in out Object;
         Item_Name  : in Identifier;
         Item       : in Any;
         Item_Flags : in Flags);
    procedure Add_Item
        (Self      : in out Object;
         Item       : in NamedValue);

--I Status free (); -- unneeded
--I Status free_memory (); -- unneeded

--I Status get_count (
--I     out longcount // number of entries in the list
--I );
    procedure Get_Count
        (Self : in Object;
         Count : out CORBA.Long);

--I };
private
    ... implementation defined ...;

end CORBA.NVList;
```

23.8.8 Package CORBA.Object

```
with CORBA.ImplementationDef;
with CORBA.InterfaceDef;
with CORBA.Context;
with CORBA.NVList;
with CORBA.Request;
--I interface Object {
package CORBA.Object is

    type Ref is tagged private;

    function To_Any(From : in Ref) return Any;
    function To_Ref(From : in Any) return Ref;

--I ImplementationDef get_implementation();
    function Get_Implementation(Self : in Ref)
        return CORBA.ImplementationDef.Ref;

--I InterfaceDef get_interface();
    function Get_Interface(Self : in Ref)
```

```

        return CORBA.InterfaceDef.Ref;

--I boolean is_nil();
function Is_Nil(Self : in Ref) return Boolean;
function Is_Null(Self : in Ref) return Boolean renames Is_Nil;

--I Object duplicate();
-- use assignment

--I void release();
procedure Release(Self : in out Ref);

--I Status create_request(
--I     in Context      ctx,
--I     in Identifier   operation,
--I     in NVList       arg_list,
--I     inout NamedValue result,
--I     out Request      request,
--I     in Flags        req_flags
--I );
procedure Create_Request
(Self      : in Ref;
Ctx       : in CORBA.Context.Object;
Operation : in Identifier;
Arg_list  : in CORBA.NVList.Object;
Result    : in out NamedValue;
Request   : out CORBA.Request.Object;
Req_Flags : in Flags;
Returns   : out Status);

private
... implementation defined ...
end CORBA.Object;

```

23.8.9 Package CORBA.ORB

```

-- Converting Object References to Strings
-- interface ORB {
with CORBA.NVList;
with CORBA.OperationDef;
with CORBA.Object;
with CORBA.Context;
with CORBA.Sequences;
with CORBA.BOA;
package CORBA.ORB is

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

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

--     Status create_list (

```

```

--      in long    count,
--      out NVListnew_list
--    );
--  procedure Create_List
--    (Count      : in      CORBA.Long;
--     New_List   : out CORBA.NVList.Object);

--    Status create_operation_list (
--      in OperationDefoper,
--      out NVListnew_list
--    );
--  procedure Create_Operation_List
--    (Oper       : in      CORBA.OperationDef.Ref;
--     New_List   : out CORBA.NVList.Object);

--    Status get_default_context (out Context ctx);
--  function Get_Default_Context return CORBA.Context.Object;

end CORBA.ORB;

```

23.8.10 Package CORBA.Principal

```

--I Interface Principal{};
package CORBA.Principal is

    type Object is private;

    function To_Any (From : in Object) return Any;
    function From_Any(From : in Any) return Object;

    function Is_Principal (Item : Any) return Boolean;

    -- implementations may add operations

private
    ... implementation defined ...
end CORBA.Principal;

```

23.8.11 Package CORBA.Request

```

--Request Routines
--I interface Request {
package CORBA.Request is

    type Object is private;

--I Status add_arg (
--I      in Identifier    name,           // argument name
--I      in TypeCode      arg_type,       // argument datatype
--I      in void           *value,        // argument value to be added
--I      in long           len,           // length/count of argument value
--I      in Flags          arg_flags      // argument flags
--I    );
--  procedure Add_Arg
--    (Self  : in out Object;
--     Arg   : in      NamedValue);

--I Status invoke (
--I      in Flags invoke_flags // invocation flags

```

```

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

--I Status delete ();
  procedure Delete
    (Self : in out Object);

--I Status send (
--I   in Flags      invoke_flags // invocation flags
--I );
  procedure Send
    (Self      : in out Object;
     Invoke_Flags : in      Flags);

--I Status get_response (
--I   in Flags      response_flags // response flags
--I );
  procedure Get_Response
    (Self      : in out Object;
     Response_Flags : in      Flags);

--I };
private
  ... implementation defined ...
end CORBA.Request;

```

23.8.12 Package *CORBA.Sequences*

```

package CORBA.Sequences is

-----
--
-- CORBA.Sequences is the parent of the bounded and unbounded sequence
-- packages. Some exceptions and types common to both are declared here
-- (following the structure of Ada.Strings).
--
-- Length_Error is raised when sequence lengths are exceeded.
-- Pattern_Error is raised when a null pattern string is passed.
-- Index_Error is raised when indexes are out of range.
--
-----

  Length_Error, Pattern_Error, Index_Error : exception;

  type Alignment is (Left, Right, Center);
  type Truncation is (Left, Right, Error);
  type Membership is (Inside, Outside);
  type Direction is (Forward, Backward);

  type Trim_End is (Left, Right, Both);

end CORBA.Sequences;

```

23.8.13 Package *CORBA.Sequences.Bounded*

```

-----
--
-- This package provides the definitions required by the IDL-to-Ada
-- mapping specification for bounded sequences.
-- This package is instantiated for each IDL bounded sequence type.
-- This package defines the sequence type and the operations upon it.
-- This package is modeled after Ada.Strings.
--
-- Most query operations are not usable until the sequence object has
-- been initialized through an assignment.
--
-- Value semantics apply to assignment, that is, assignment of a sequence
-- value to a sequence object yields a copy of the value.
--
-- The exception INDEX_ERROR is raised when indexes are not in the range
-- of the object being manipulated.
--
-- The exception CONSTRAINT_ERROR is raised when objects that have not
-- been initialized or assigned to are manipulated.
--
-----

generic

  type Element is private;
  Max : Positive;    -- Maximum length of the bounded sequence

package CORBA.Sequences.Bounded is

  Max_Length : constant Positive := Max;

  type Element_Array is array (Positive range <>) of Element;

  Null_Element_Array : Element_Array(1..0);

  type Sequence is private;

  Null_Sequence : constant Sequence;

  subtype Length_Range is Natural range 0 .. Max_Length;

  function Length (Source : in Sequence)
    return Length_Range;

  type Element_Array_Access is access all Element_Array;
  procedure Free(X : in out Element_Array_Access);

  -----
  -- Conversion, Concatenation, and Selection Functions --
  -----

  function To_Sequence
    (Source : in Element_Array;
     Drop   : in Truncation := Error)
    return Sequence;

  function To_Sequence
    (Length : in Length_Range)
    return   Sequence;

```



```

function To_Element_Array (Source : in Sequence)
    return Element_Array;

function Append
    (Left, Right : in Sequence;
     Drop       : in Truncation := Error)
    return      Sequence;

function Append
    (Left  : in Sequence;
     Right : in Element_Array;
     Drop  : in Truncation := Error)
    return Sequence;

function Append
    (Left  : in Element_Array;
     Right : in Sequence;
     Drop  : in Truncation := Error)
    return Sequence;

function Append
    (Left  : in Sequence;
     Right : in Element;
     Drop  : in Truncation := Error)
    return Sequence;

function Append
    (Left  : in Element;
     Right : in Sequence;
     Drop  : in Truncation := Error)
    return Sequence;

procedure Append
    (Source  : in out Sequence;
     New_Item : in Sequence;
     Drop    : in Truncation := Error);

procedure Append
    (Source  : in out Sequence;
     New_Item : in Element_Array;
     Drop    : in Truncation := Error);

procedure Append
    (Source  : in out Sequence;
     New_Item : in Element;
     Drop    : in Truncation := Error);

function "&" (Left, Right : in Sequence)
    return Sequence;

function "&"
    (Left  : in Sequence;
     Right : in Element_Array)
    return Sequence;

function "&"
    (Left  : in Element_Array;
     Right : in Sequence)
    return Sequence;

```

```

function "&"
    (Left  : in Sequence;
     Right : in Element)
    return Sequence;

function "&"
    (Left  : in Element;
     Right : in Sequence)
    return Sequence;

function Element_Of
    (Source : in Sequence;
     Index  : in Positive)
    return Element;

procedure Replace_Element
    (Source : in out Sequence;
     Index  : in Positive;
     By     : in Element);

function Slice
    (Source : in Sequence;
     Low    : in Positive;
     High   : in Natural)
    return Element_Array;

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

function "="
    (Left  : in Sequence;
     Right : in Element_Array)
    return Boolean;

function "="
    (Left  : in Element_Array;
     Right : in Sequence)
    return Boolean;

-----
-- Search functions --
-----

function Index
    (Source  : in Sequence;
     Pattern : in Element_Array;
     Going   : in Direction := Forward)
    return Natural;

function Count
    (Source  : in Sequence;
     Pattern : in Element_Array)
    return Natural;

-----
-- Sequence transformation subprograms --
-----

function Replace_Slice
    (Source : in Sequence;
     Low    : in Positive;

```

```

        High   : in Natural;
        By     : in Element_Array;
        Drop   : in Truncation := Error)
    return Sequence;

procedure Replace_Slice
    (Source   : in out Sequence;
     Low      : in Positive;
     High     : in Natural;
     By       : in Element_Array;
     Drop     : in Truncation := Error);

function Insert
    (Source   : in Sequence;
     Before   : in Positive;
     New_Item : in Element_Array;
     Drop     : in Truncation := Error)
    return Sequence;

procedure Insert
    (Source   : in out Sequence;
     Before   : in Positive;
     New_Item : in Element_Array;
     Drop     : in Truncation := Error);

function Overwrite
    (Source   : in Sequence;
     Position : in Positive;
     New_Item : in Element_Array;
     Drop     : in Truncation := Error)
    return Sequence;

procedure Overwrite
    (Source   : in out Sequence;
     Position : in Positive;
     New_Item : in Element_Array;
     Drop     : in Truncation := Error);

function Delete
    (Source   : in Sequence;
     From     : in Positive;
     Through  : in Natural)
    return Sequence;

procedure Delete
    (Source   : in out Sequence;
     From     : in Positive;
     Through  : in Natural);

-----
-- Sequence selector subprograms --
-----

function Head
    (Source : in Sequence;
     Count  : in Natural;
     Pad    : in Element;
     Drop   : in Truncation := Error)
    return Sequence;

procedure Head

```

```

        (Source : in out Sequence;
         Count  : in Natural;
         Pad    : in Element;
         Drop   : in Truncation := Error);

function Tail
  (Source : in Sequence;
   Count  : in Natural;
   Pad    : in Element;
   Drop   : in Truncation := Error)
return Sequence;

procedure Tail
  (Source : in out Sequence;
   Count  : in Natural;
   Pad    : in Element;
   Drop   : in Truncation := Error);

-----
-- Sequence constructor subprograms --
-----

function "*"
  (Left  : in Natural;
   Right : in Element)
return Sequence;

function "*"
  (Left  : in Natural;
   Right : in Element_Array)
return Sequence;

function "*"
  (Left  : in Natural;
   Right : in Sequence)
return Sequence;

function Replicate
  (Count : in Natural;
   Item  : in Element;
   Drop  : in Truncation := Error)
return Sequence;

function Replicate
  (Count : in Natural;
   Item  : in Element_Array;
   Drop  : in Truncation := Error)
return Sequence;

function Replicate
  (Count : in Natural;
   Item  : in Sequence;
   Drop  : in Truncation := Error)
return Sequence;

private

  ... implementation defined ...

end CORBA.Sequences.Bounded;

```

23.8.14 Package *CORBA.Sequences.Unbounded*

```

-----
-- This package provides the definitions required by the IDL-to-Ada
-- mapping specification for unbounded sequences.
-- This package is instantiated for each IDL unbounded sequence type.
-- This package defines the sequence type and the operations upon it.
-- This package is modeled after Ada.Strings.
--
-- Most query operations are not usable until the sequence object has
-- been initialized through an assignment.
--
-- Value semantics apply to assignment, that is, assignment of a sequence
-- value to a sequence object yields a copy of the value.
--
-- The exception INDEX_ERROR is raised when indexes are not in the range
-- of the object being manipulated.
--
-- The exception CONSTRAINT_ERROR is raised when objects that have not
-- been initialized or assigned to are manipulated.
--
-----

generic

  type Element is private;

package CORBA.Sequences.Unbounded is

  type Element_Array is array (integer range <>) of Element;

  Null_Element_Array : Element_Array(1..0);

  type Sequence is private;

  Null_Sequence : constant Sequence;

  function Length (Source : in Sequence)
    return Natural;

  type Element_Array_Access is access all Element_Array;
  procedure Free(X : in out Element_Array_Access);

  -----
  -- Conversion, Concatenation, and Selection Functions --
  -----

  function To_Sequence
    (Source : in Element_Array)
    return Sequence;

  function To_Sequence
    (Length : in Natural)
    return Sequence;

  function To_Element_Array (Source : in Sequence)
    return Element_Array;

  procedure Append
    (Source : in out Sequence;

```

```

        New_Item : in Sequence);

procedure Append
    (Source      : in out Sequence;
     New_Item    : in Element_Array);

procedure Append
    (Source      : in out Sequence;
     New_Item    : in Element);

function "&" (Left, Right : in Sequence)
    return Sequence;

function "&"
    (Left  : in Sequence;
     Right : in Element_Array)
    return Sequence;

function "&"
    (Left  : in Element_Array;
     Right : in Sequence)
    return Sequence;

function "&"
    (Left  : in Sequence;
     Right : in Element)
    return Sequence;

function "&"
    (Left  : in Element;
     Right : in Sequence)
    return Sequence;

function Element_Of
    (Source : in Sequence;
     Index  : in Positive)
    return Element;

procedure Replace_Element
    (Source : in out Sequence;
     Index  : in Positive;
     By     : in Element);

function Slice
    (Source : in Sequence;
     Low    : in Positive;
     High   : in Natural)
    return Element_Array;

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

function "="
    (Left  : in Element_Array;
     Right : in Sequence)
    return Boolean;

function "="
    (Left  : in Sequence;
     Right : in Element_Array)
    return Boolean;

```

```

-----
-- Search functions --
-----

function Index
    (Source      : in Sequence;
     Pattern     : in Element_Array;
     Going       : in Direction := Forward)
    return      Natural;

function Count
    (Source      : in Sequence;
     Pattern     : in Element_Array)
    return      Natural;

-----
-- Sequence transformation subprograms --
-----

function Replace_Slice
    (Source      : in Sequence;
     Low         : in Positive;
     High        : in Natural;
     By          : in Element_Array)
    return      Sequence;

procedure Replace_Slice
    (Source      : in out Sequence;
     Low         : in Positive;
     High        : in Natural;
     By          : in Element_Array);

function Insert
    (Source      : in Sequence;
     Before      : in Positive;
     New_Item    : in Element_Array)
    return      Sequence;

procedure Insert
    (Source      : in out Sequence;
     Before      : in Positive;
     New_Item    : in Element_Array);

function Overwrite
    (Source      : in Sequence;
     Position    : in Positive;
     New_Item    : in Element_Array)
    return      Sequence;

procedure Overwrite
    (Source      : in out Sequence;
     Position    : in Positive;
     New_Item    : in Element_Array);

function Delete
    (Source      : in Sequence;
     From        : in Positive;
     Through     : in Natural)
    return      Sequence;

```

```

procedure Delete
    (Source : in out Sequence;
     From   : in Positive;
     Through : in Natural);

-----
-- Sequence selector subprograms --
-----
function Head
    (Source : in Sequence;
     Count  : in Natural;
     Pad    : in Element)
    return Sequence;

procedure Head
    (Source : in out Sequence;
     Count  : in Natural;
     Pad    : in Element);

function Tail
    (Source : in Sequence;
     Count  : in Natural;
     Pad    : in Element)
    return Sequence;

procedure Tail
    (Source : in out Sequence;
     Count  : in Natural;
     Pad    : in Element);

-----
-- Sequence constructor subprograms --
-----

function "*"
    (Left  : in Natural;
     Right : in Element)
    return Sequence;

function "*"
    (Left  : in Natural;
     Right : in Element_Array)
    return Sequence;

function "*"
    (Left  : in Natural;
     Right : in Sequence)
    return Sequence;

private

    ... implementation defined ...
end CORBA.Sequences.Unbounded;

```


23.9 Glossary of Ada 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.

