Ada Language Mapping Specification

Version 1.2: October 2001
New Edition: June 1999

PATENT

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

NOTICE

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

WHILE THE INFORMATION IN THIS PUBLICATION IS BELIEVED TO BE ACCURATE, THE OBJECT MANAGEMENT GROUP AND THE COMPANIES LISTED ABOVE MAKE NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS MATERIAL INCLUDING, BUT NOT LIMITED TO ANY WARRANTY OF TITLE OR OWNERSHIP, IMPLIED WARRANTY OF MERCHANTABILITY OR WARRANTY OF FITNESS FOR PARTICULAR PURPOSE OR USE. In no event shall The Object Management Group or any of the companies listed above be liable for errors contained herein or for indirect, incidental, special, consequential, reliance or cover damages, including loss of profits, revenue, data or use, incurred by any user or any third party. The copyright holders listed above acknowledge that the Object Management Group (acting itself or through its designees) is and shall at all times be the sole entity that may authorize developers, suppliers and sellers of computer software to use certification marks, trademarks or other special designations to indicate compliance with these materials.

ISSUE REPORTING

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

Contents

Contents

Contents

Contents

Preface

About the Object Management Group

The Object Management Group, Inc. (OMG) is an international organization supported by over 800 members, including information system vendors, software developers and users. Founded in 1989, the OMG promotes the theory and practice of object-oriented technology in software development. The organization's charter includes the establishment of industry guidelines and object management specifications to provide a common framework for application development. Primary goals are the reusability, portability, and interoperability of object-based software in distributed, heterogeneous environments. Conformance to these specifications will make it possible to develop a heterogeneous applications environment across all major hardware platforms and operating systems.

OMG's objectives are to foster the growth of object technology and influence its direction by establishing the Object Management Architecture (OMA). The OMA provides the conceptual infrastructure upon which all OMG specifications are based.

What is CORBA?

The Common Object Request Broker Architecture (CORBA), is the Object Management Group's answer to the need for interoperability among the rapidly proliferating number of hardware and software products available today. Simply stated, CORBA allows applications to communicate with one another no matter where they are located or who has designed them. CORBA 1.1 was introduced in 1991 by Object Management Group (OMG) and defined the Interface Definition Language (IDL) and the Application Programming Interfaces (API) that enable client/server object interaction within a specific implementation of an Object Request Broker (ORB). CORBA 2.0, adopted in December of 1994, defines true interoperability by specifying how ORBs from different vendors can interoperate.

About CORBA Language Mapping Specifications

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

- Ada
- C
- C++
- COBOL
- IDL Script
- IDL to Java
- Java to IDL
- Lisp
- Python
- Smalltalk

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

Alignment with CORBA

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

Language Mapping	**Aligned with CORBA version**
Ada	CORBA 2.3 + components
C	CORBA 2.1
C++	CORBA 2.3
COBOL	CORBA 2.1
IDL Script	CORBA 2.3
IDL to Java	CORBA 2.3
Java to IDL	CORBA 2.3
Lisp	CORBA 2.3
Python	CORBA 2.3
Smalltalk	CORBA 2.0

Definition of CORBA Compliance

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

their ORB must comply with the OMG IDL to C++ binding specified in this manual.

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

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

The *CORBA* specifications are divided into these volumes:

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

Acknowledgements

The following company submitted the specification that was approved by the Object Management Group to become the Ada Language Mapping specification:

- Objective Interface Systems, Inc.

Overview 1

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 (a.k.a. CORBA/IIOP 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/IIOP 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.

Contents

This chapter contains the following sections.

1.1 General Requirements

1.1.1 Ada Implementation Requirements

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

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

1.1.2 Calling Convention

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

1.1.3 Memory Management

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

1.1.4 Tasking

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

1.1.5 Ada Type Size Requirements

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

1.2 Mapping Summary

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

Table 1-1 Summary of IDL Constructs to Ada Constructs

IDL construct	Ada construct
`Source file`	`Library package`
`Module`	`Package (Child Package if nested)`
`Interface`	`Package with Tagged Type (Child Package if nested)`
`Operation`	`Primitive Subprogram`
`Attribute`	`"Set_attribute" and "Get_attribute" subprograms`

`Inheritance:`
`  Single`
`  Multiple`
	
	`Tagged Type Inheritance`
	`Tagged Type Inheritance for first parent; cover functions with explicit widening and narrowing for subsequent parents`

`Data types`	`Ada types`
`Exception`	`Exception and record type`

1.3 Interfaces and Tagged Types

1.3.1 Client Side

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

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

1.3.2 Forward Declarations

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

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

1.3.3 Server Side

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

1.4 Operations

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

1.5 Attributes

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

1.6 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; that is, an interface with a parent is mapped to a tagged type that is derived from the tagged type mapped from the parent. The definitions of types, constants, and exceptions in the parent package are renamed or subtyped so that they are also "inherited" in accordance with the IDL semantics.

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

1.7 Data Types

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

Table 1-2 Summary of Mapping Types

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

1.8 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. Implementers must provide a function that returns the exception members from the Ada-provided **`Exception_Occurrence`** for each exception type.

1.9 Names and Scoping

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

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

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

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

1.10 New and Changed Features of the Ada Mapping

This section presents an overview and rationale for the new features of the IDL to Ada mapping. The following chapters revise the Ada Language Mapping specification (now published by the OMG as stand-alone volumes). Change bars in the following chapters indicate substantive (as opposed to organizational) changes from the currently adopted specification (OMG TC document number ptc/99-03-11).

Change bars in the following chapters highlight changes from the initial submission (orbos/99-07-06).

1.10.1 Helper Packages

The current mapping of interfaces to Ada requires the generation of a **To_Ref** function in the interface package, which supports widening and narrowing of object references. In addition, the functions supporting conversion of an object reference to and from type **Any** must be statically defined in the **CORBA.Object package**. Because Ada’s rules require overriding of functions that return the type being derived from, the **From_Any** function had to be generated for every interface package, regardless of whether the developer had asked the IDL compiler for the generated functions supporting type **Any**. Other language mappings, when faced with this cluttering of the mapped interface, and to facilitate reverse mappings, have relegated some of these supporting operations to “helper modules.”

This revision of the mapping adopts this strategy. The mapping of interfaces now requires the generation of three packages:

1. The interface package. As in the present mapping, this package will define the Ref type and specify its primitive operations mapped from the attribute accessors and operations of the IDL interface.

2. The interface implementation package. As in the present mapping, this package will define the servant **Object** type and the specification of its primitive operations, which are to be implemented by the developer.

3. An interface helper package. This package will be a child package of the interface package with extended name **.Helper**. It will contain:
   - The **To_Ref** function that supports widening and narrowing of object references.
   - The interface **TypeCode** and supporting conversion functions: **To_Any** and **From_Any**.

This packaging pattern is also used for the mapping of new IDL constructs, as seen below.

1.10.2 Value Types

As stated in the CORBA standard:

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

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

The Ada mapping for value types provides all of these semantics, and is complementary to the Ada mapping for Object References.

1.10.2.1 Basic Mapping

A value type is mapped to three packages in Ada.

1. A value interface package. This package is a child of the package mapped from the IDL scope declaring the value. The name of the package is the value identifier appended to the parent package name. This package contains:
   - A **`Value_Ref`** type that represents the mapping of the value type. The **`Value_Ref`** type provides reference counting and "smart pointer" semantics, similar to those provided by the **`Ref`** type that is the mapping of IDL interfaces. These reference semantics provide the required support of arbitrary graphs. The **`Value_Ref`** type "points to" a **`Value_Impl.Object`**.
   - Accessor functions and procedures for the public state members of a value.
   - Functions and procedures mapped from the operations on the value type. The signatures of these operations are consistent with those mapped from interface operations. Although these "primitive operations" are simply pass-throughs to the actual implementations in the **`Value_Impl`** package, they provide the required inheritance semantics for derived value types.
   - Functions returning a **`Value_Ref`** mapped from the initializers specified in IDL.
   - A "null value" constant that represents the value of uninitialized or "null" **`Value_Ref`** variables.
2. A value implementation package. This package is a child of the value interface package and has the name extension **`.Value_Impl`**. This package contains:
   - An **`Object`** type that represents the concrete implementation of the value type. The **`Value_Impl.Object`** type contains components for each of the public and private state members of the value type.
   - The **`Object_Ptr`** type, which is a general access to the **`Object`** class.

- Functions and procedures mapped from the operations on the value type. The signatures of these operations are consistent with those in Impl packages mapped from interface operations. The bodies of these operations are implemented by the CORBA developers.
- Factory functions returning an **`Object_Ptr`**. The bodies of these functions are implemented by the CORBA developers.

Actually, only the package specification[1] of the value implementation package needs to be generated; the body of the package can only be written by the developer.

3. A value helper package. The value helper package is a child of the value interface package and has the name extension **`.Helper`**. This package contains:
   - The definition of the **`TypeCode`** constant for the value type, and the **`From_Any`** and **`To_Any`** functions for the value type.
   - Any necessary widening conversion functions. These are only needed for abstract values (and abstract interfaces).

For example, the following IDL:

```
// IDL - ExampleA.idl
module ExampleA {

    typedef sequence<unsigned long> WeightSeq;

    valuetype WeightedBinaryTree {
        public long weight;
        private WeightedBinaryTree left;
        private WeightedBinaryTree right;
        factory createWBT(in long w);
        WeightSeq preOrder();
        WeightSeq postOrder();
    };

};
```

maps to:

```
-- Ada - examplea.ads
with CORBA.Unsigned_Long_Unbounded;
package ExampleA is

    type WeightSeq is ...

end ExampleA;
```

1. Ada formalizes the separation of declaration of a package's interface, its specification, from a package's implementation, the body, by requiring different syntax for the two. Those more familiar with C++ might best think of this as an enforced separation of "pure" header files from the files containing the actual implementation.

```
-- Ada - examplea-weightedbinarytree.ads
with CORBA.Value;
package ExampleA.WeightedBinaryTree is

    type Value_Ref is new CORBA.Value.Base with null record;
    Null_Value : constant Value_Ref;

    function  Get_weight(Self : Value_Ref) return CORBA.Long;
    procedure Set_weight(Self : Value_Ref; To : CORBA.Long);

    function createWBT(w : in CORBA.Long) return Value_Ref;

    function preOrder (Self: Value_Ref)
        return ExampleA.WeightSeq;
    function postOrder (Self: Value_Ref)
        return ExampleA.WeightSeq;

end ExampleA.WeightedBinaryTree;

-- Ada - examplea-weightedbinarytree-value_impl.ads
with CORBA.Value;
package ExampleA.WeightedBinaryTree.Value_Impl is

    type Object is new CORBA.Value.Impl_Base with record
        weight : CORBA.Long;
        left   : Value_Ref;
        right  : Value_Ref;
    end record;
    type Object_Ptr is access all Object'class;

    function preOrder (Self: access Object) return WeightSeq;
    function postOrder (Self: access Object)
        return WeightSeq;

    function wreateWBT(W : in CORBA.Long) return Object_Ptr;

end ExampleA.WeightedBinaryTree.Value_Impl;

-- Ada - examplea-weightedbinarytree-helper.ads
package ExampleA.WeightedBinaryTree.Helper is

    function To_Any (From : in Value_Ref) return CORBA.Any;

    function From_Any (From : in CORBA.Any) return Value_Ref;

    TC_WeightedBinaryTree : constant CORBA.TypeCode.Object;

end ExampleA.WeightedBinaryTree.Helper;
```

1.10.2.2 Abstract vs. Concrete

Again, from the *CORBA/IIOP Specification*:

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

The Ada mapping for abstract values differs only slightly from that for concrete values: the **`Value_Impl.Object`** type and the enclosing **`Value_Impl`** package are not needed. This prevents instance of the abstract type from being created.

Note the **`Value_Ref`** type and associated operations are not abstract. This allows the widening of a concrete value type to an abstract value type (for example, in order to pass as the actual for a formal parameter that is of the abstract value type). This mapping also allows the operations of the abstract value type to be invoked on a reference obtained in this way.

In order to support widening from concrete value types that may inherit from the abstract value type, the **`.Helper`** child package will contain a **`To_Abstract_Value_Ref`** function that will widen a reference to any concrete or abstract value that inherits from it to a valid reference of the abstract type.

For example, the following IDL:

```
// IDL
module CORBA {
   abstract valuetype CustomMarshal {
      void marshal (in DataOutputStream ostream);
      void unmarshal (in DataInputStream istream);
   };
};
```

maps to:

```
-- Ada - corba-custommarshal.ads
with CORBA.Value;
with CORBA.DataOutputStream;
with CORBA.DataInputStream;
package CORBA.CustomMarshal is

    type Abstract_Value_Ref is
        new CORBA.Value.Base with null record;

    procedure marshal
        (Self    :    Abstract_Value_Ref;
```

```
            Ostream : in
                    CORBA.DataOutputStream.Abstract_Value_Ref);
        procedure unmarshal
            (Self    :    Abstract_Value_Ref;
             Istream : in
                    CORBA.DataInputStream.Abstract_Value_Ref);
end CORBA.CustomMarshal;

package CORBA.CustomMarshal.Helper is

    function To_Any (From : in Abstract_Value_Ref)
        return CORBA.Any;

    function From_Any (From : in CORBA.Any)
        return Abstract_Value_Ref;

    TC_CustomMarshal : constant CORBA.TypeCode.Object;

    function To_Abstract_Value_Ref
        (From : in CORBA.Value.Base'CLASS)
            return Abstract_Value_Ref;

end CORBA.CustomMarshal.Helper;
```

1.10.2.3 *Inheriting from Stateful Value Types*

Value types may (singly) inherit from another concrete value type. The Ada mapping for inheritance in this case provides for both interface inheritance (subtyping) and implementation inheritance (subclassing). Both the **`Value_Ref`** type and the **`Value_Impl.Object`** type for the inherited type use Ada's tagged type derivation to inherit the state members and operations from the corresponding types mapped from the parent concrete value type.

The derived value type may be widened to the parent value type through Ada's view conversion syntax.

1.10.2.4 *Inheriting from Abstract Value Types*

Value types (stateful or abstract) may also inherit from one or more abstract value types. In this situation, there is no need for implementation inheritance (there can be no implementation of the abstract value type), but there is the need for interface inheritance from multiple entities. Ada does not directly support multiple inheritance. Instances of concrete value types may also be widened to one of their abstract value ancestors (for example, for use in a parameter that has formal type of the abstract value type).

The Ada mapping for value type inheritance from abstract value types is similar to that used for interfaces that multiply inheritance from other interfaces. The operations inherited from the abstract value type are "copied down" into both the value interface package and the value implementation package.

Instances of abstract or stateful value types may be widened to inherited abstract value types through the **`To_Abstract_Value_Ref`** function defined in the value helper package of the abstract parent value type.

1.10.2.5 *Values Supporting Interfaces*

Value types may also "support" interfaces, a relationship that is similar to but not exactly the same as the interface inheritance (subtyping) relationship specified as the semantics for "inheritance" between interfaces. Abstract value may support multiple interfaces, while stateful values may only support one. From the *CORBA/IIOP Specification*:

> *"They [value types] can be used to specify the state of an object implementation; that is, they can support an interface."*

An instance of a value supporting an interface may be passed as a parameter of an operation that has a formal type of the interface (substitutability). The instance being passed must have previously been activated with a POA (an instance of a value type that supports an interface can be "widened" to that interface).

The Ada value interface package mapped from an IDL value type supporting an interface includes "copied down" operations mapped from the operations on the supported interface (and all of its ancestors). The value implementation package also includes subprograms mapped from these operations. The developer writes the bodies of these operations for local calls on the value interface package.

In addition, the value helper package (for each supported interface):

- Defines **`Servant`** and **`Servant_Ref`** types that "wrap" the **`Value_Impl.Object`** type.
- Includes a **`To_Servant`** function that returns a **`Servant_Ref`** instance for an instance of the value's **`Value_Impl.Object`** type. Once a **`Servant_Ref`** has been obtained, it can be activated with a POA.

This satisfies the substitutability requirement for values that support interfaces.

For example, the following IDL:

```
// IDL
module ExampleB {

   interface Printer{
      typedef sequence<unsigned long> ULongSeq;
      void print(in ULongSeq data);
   };

   valuetype WeightedBinaryTree supports Printer {
      public long weight;
      private WeightedBinaryTree left;
      private WeightedBinaryTree right;
      factory createWBT(in long w);
      ULongSeq preOrder();
      ULongSeq postOrder();
   };

};
```

maps to:

```
-- Ada - exampleb-printer.ads
with CORBA.Object;
with CORBA.Unsigned_Long_Unbounded;
package Exampleb.Printer is

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

    type ULongSeq is ...

    procedure Print(Self: Ref; Data: in ULongSeq);

end Exampleb.Printer;

-- Ada - exampleb-weightedbinarytree.ads
with CORBA.Value;
with CORBA.Value;
with Exampleb.Printer;
package Exampleb.WeightedBinaryTree is

    type Value_Ref is new CORBA.Value.Base with null record;
    Null_Value : constant Value_Ref;

    function  Get_weight(Self : Value_Ref) return CORBA.Long;
    procedure Set_weight(Self : Value_Ref; To : CORBA.Long);

    function createWBT(w : in CORBA.Long) return Value_Ref;

    function preOrder (Self: Value_Ref)
        return ExampleB.Printer.ULongSeq;
```

```
    function postOrder (Self: Value_Ref)
        return ExampleB.Printer.ULongSeq;

    procedure print
        (Self :    Value_Ref;
         data : in Printer.ULongSeq);

end Exampleb.WeightedBinaryTree;

-- Ada - exampleb-weightedbinarytree-value_impl.ads
package ExampleB.WeightedBinaryTree.Value_Impl is

    type Object is new CORBA.Value.Impl_Base with record
        weight : CORBA.Long;
        left   : Value_Ref;
        right  : Value_Ref;
    end record;
    type Object_Ptr is access all Object'CLASS;

    function createWBT(w : in CORBA.Long) return Object_Ptr;

    function preOrder (Self: access Object)
        return ExampleB.Printer.ULongSeq;
    function postOrder (Self: access Object)
        return ExampleB.Printer.ULongSeq;

    procedure print
        (Self : access Object;
         Data : in     ExampleB.Printer.ULongSeq);

end ExampleB.WeightedBinaryTree.Value_Impl;

-- Ada - example-weightedbinarytree-helper.ads
with ExampleB.WeightedBinaryTree.Value_Impl;
with PortableServer;
package ExampleB.WeightedBinaryTree.Helper is

    function To_Any (From : in Value_Ref) return CORBA.Any;

    function From_Any (From : in CORBA.Any) return Value_Ref;

    TC_WeightedBinaryTree : constant CORBA.TypeCode.Object;

    type Servant
       (Value: access
        ExampleB.WeightedBinaryTree.Value_Impl.Object'CLASS)
      is
        new PortableServer.Servant with null record;
    type Servant_Ref is access all Servant'CLASS;
```

```
        function To_Servant
            (Self : access
            ExampleB.WeightedBinaryTree.Value_Impl.Object'CLASS)
            return Servant_Ref;

end ExampleB.WeightedBinaryTree.Helper;
```

1.10.2.6 Values Supporting Abstract Interfaces

Abstract or stateful values may support multiple abstract interfaces. See Sectio n1.10.4, “Abstract Interfaces,” on page 1-19 for the mapping of abstract interfaces. Support of an abstract interface is similar to support of a non-abstract interface:

- Operations from the supported abstract interface are “copied down” and mapped to subprograms in both the value interface package and, for stateful value types, the value implementation package.

However an instance of a value type supporting an abstract interface cannot be registered as a servant, since there are no servants for abstract interfaces. Therefore, the extra types and operations needed in the value helper package for support of concrete interfaces is not needed.

The widening of an instance of a value type to an abstract interface reference is provided by the **`To_Abstract_Ref`** function in the helper package of the mapped abstract interface.

1.10.2.7 Base Types for Values

This revision of the mapping adds the package **`CORBA.Value`** to the packages that must be provided by a conforming implementation. This package must contain the following definitions:

```
-- Ada
with CORBA.AbstractBase;
package CORBA.Value is

    type Base is abstract new CORBA.AbstractBase.Ref
        with null record;
    type Impl_Base is abstract  tagged limited private;

end CORBA.Value;
```

Conceptually, instances of the **`CORBA.Value.Base`** type point to instances of the **`CORBA.Value.Impl_Base`** type. Conforming implementations must provide pointer semantics and reference counting, but these semantics will probably be directly inherited from the parent types.

1.10.2.8 Forward Declaration of Values

There is provision in IDL for the introduction of a value type, so that the value type may be used before it is fully defined. In this way, systems of value types may recursively refer to each other. A similar provision is present for interface types (although most uses of forward declaration of interfaces seem to be so that typing design flaws do not have to be fixed.)

Ada restricts definition circularities to a greater extent than many other languages to ensure stronger type safety[2]. In particular, it is an error for packages to "with" each other, either directly or indirectly. If all IDL forward declarations are resolved to the corresponding "real" declarations and mapped to Ada, there are many legal IDL files that result in illegal circular dependencies among Ada packages. Therefore, the mapping of IDL value type to Ada requires a special mapping for value forward declarations. Fortunately, the procedures for mapping forward interface declarations are readily adaptable to value forward declarations.

A new generic package, **`CORBA.Value.Forward`**, is defined to support this:

```
-- Ada
generic
package CORBA.Value.Forward is

    type Value_Ref is new CORBA.Value.Base with null record;

    generic
        type Ref_Type is new CORBA.Value.Base with private;
    package Convert is

        function From_Forward (The_Forward : in Value_Ref)
            return Ref_Type;
        function To_Ref       (The_Forward : in Value_Ref)
            return Ref_Type renames From_Forward;

        function To_Forward (The_Ref : in Ref_Type)
            return Value_Ref;

    end Convert;

end CORBA.Value.Forward;
```

The mapping of a value forward declaration results in:

---

2. There is a proposal before ISO WG9 that relaxes this restriction in the Ada language and would obviate the need for the generic package described in this subsection. However, until this proposal is passed and widely supported in compilers, the present mapping must be retained.

- An instantiation of **`CORBA.Value.Forward`** at the point of the forward declaration. The instantiated package name is formed by appending "**`_Forward`**" to value type identifier.
- The type ***`value`*`_Forward.Ref_Type`**, created by this instantiation, is used as the mapping of the value type until the IDL value type is actually defined.
- In the interface package mapped from the actual value type definition, the ***`value`*`_Forward.Convert`** package is instantiated with the actual Ada **`Value_Ref`** type mapped from the value.

This series of instantiations provides convenient functions for type-safe conversions between instances of the forward **`Value_Ref`** type and the actual **`Value_Ref`** type.

1.10.2.9 Custom Marshalling

The Ada mappings for the IDL abstract value types, **CORBA::CustomMarshal**, **CORBA::DataOutputStream**, and **CORBA::DataInputStream** follow the normal Ada value type mapping rules.

1.10.3 Value Boxes

The *CORBA/IIOP Specification* states this about value boxes:

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

Value boxes may be less useful in Ada than in other languages; all CORBA-defined Ada types are automatically memory-managed. Indeed, the specification goes on to cite strings and sequences as particularly useful types to be "boxed." In Ada, these are controlled types that allocate additional memory when needed and free it after instances of these types go out of scope.

In order to support the mapping of value boxes to Ada, conforming implementations must provide an implementation of the following generic package:

```
generic
    type Boxed is private;
    type Boxed_Access is access all Boxed;
package CORBA.Value.Box is

    type Box_Ref is new CORBA.Value.Base with private;

    function Is_Null(The_Ref : Box_Ref) return Boolean;

    function Create(With_Value : in Boxed) return Box_Ref;
    function "+"   (With_Value : in Boxed) return Box_Ref
        renames Create;

    function Contents(The_Boxed : in Box_Ref)
        return Boxed_Access;
    function     "-" (The_Boxed : in Box_Ref)
        return Boxed_Access renames Contents;

    procedure Release(The_Ref : in out Box_Ref);

end CORBA.Value.Box;
```

Implementations of the **Box_Ref** type must support reference counting, “smart pointer” semantics for the boxed value.

The mapping of an IDL value box declaration consists of:

- The declaration of a general access to the type to be boxed.
- An instantiation **CORBA.Value.Box** with the type to be boxed and declared access type as actual parameters of the instantiation. The name of the instantiated package is formed by appending **_Value_Box** to the IDL identifier for the value box.
- A derivation of the **Box_Ref** type from the instantiation with name mapped from the identifier of the value box. This has the effect of introducing the type and its operations into the original name scope.

For example, the following IDL:

```
// IDL
module Example {

   valuetype LongSeq sequence<Long>;
   interface Bar {
      void doit(in LongSeq seq1);
   };
};
```

maps to:

```
-- Ada - example.ads
with CORBA.Sequences.Unbounded;
with CORBA.Value.Box;
package Example is

    type Long_Array is array(Integer range <>) of CORBA.Long;
    package IDL_SEQUENCE_Long is new
        CORBA.Sequences.Unbounded(CORBA.Long, Integer,
                                  Long_Array, "=");

    type IDL_SEQUENCE_Long_Access is
        access all IDL_SEQUENCE_Long.Sequence;
    package LongSeq_Value_Box is
        new CORBA.Value.Box(IDL_SEQUENCE_Long.Sequence,
                            IDL_SEQUENCE_Long_Access);
    type LongSeq is new LongSeq_Value_Box.Box_Ref;

end Example;

-- Ada - example-bar.ads
with CORBA.Object;
package Example.Bar is

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

    procedure Doit(Self : in Ref; seq1: in Example.LongSeq);

end Example.Bar;
```

1.10.4 Abstract Interfaces

The *CORBA/IIOP Specification* states:

> *"In many cases it may be useful to defer the determination of whether an object is passed by reference or by value until runtime. An IDL abstract interface provides this capability."*

The semantics of abstract interfaces differ in a number of ways, and the CORBA specification specifically allows different language mappings for abstract interfaces versus concrete interfaces.

1.10.4.1 Basic Mapping

The mapping for an abstract interface into Ada includes:

1. An abstract interface package. This package is a child of the package mapped from the IDL scope declaring the abstract interface. The name of the package is the abstract interface identifier appended to the parent package name. This package contains:

- An **Abstract_Ref** type that represents the mapping of the abstract interface type. The **Abstract_Ref** type contributes to the reference counting and "smart pointer" semantics of the actual interface implementation or value type implementation that it refers to.
- Functions and procedure mapped from the operations on the abstract interface type. The signatures of these operations are consistent with those mapped from interface operations. Although these "primitive operations" are simply pass-throughs to the actual implementations referred to, they provide the required inheritance semantics for derived abstract interface types and supporting value types.

2. An abstract interface helper package. The abstract interface helper package is a child of the abstract interface package and has the name extension **.Helper**. This package contains:
   - The definition of a **To_Any** function. This function must "dispatch" to encode the TypeCode and value according to the rules for encoding the contents of the value type or interface type referred to by the abstract interface.
   - A **To_Abstract_Ref** function that is capable of widening a reference to a supporting interface or value type to a reference to the abstract interface type.

Note that there is no "impl" package, since the abstract interface cannot be directly implemented.

For example, in the following IDL:

```
// IDL
module Example {

  exception e{};

  interface Marker {};

  abstract interface Base {
    void baseOp();
  };

  interface Extended: Base, Marker {
    long method (in long arg) raises (e);
    attribute long assignable;
    readonly attribute long nonassignable;
  };
};
```

the abstract interface, **Base**, is mapped to:

```
-- Ada - example-base.ads
with CORBA.AbstractBase;
package Example.Base is

    type Abstract_Ref is new CORBA.AbstractBase.Ref
        with null record;

    procedure BaseOp(Self : Abstract_Ref);

end Example.Base;
```

```
-- Ada - example.base.helper
package Example.Base.Helper is

    function To_Any (From : in Abstract_Ref)
        return CORBA.Any;

    function To_Abstract_Ref
        (From : in CORBA.AbstractBase.Ref'CLASS)
        return Abstract_Ref;

end Example.Base.Helper;
```

1.10.4.2 Abstract Interfaces Inheriting from Abstract Interfaces

Abstract interfaces may only inherit from other abstract interfaces, but they may inherit from multiple ones. The Ada mapping of this inheritance simply requires the "copying down" of the inherited operations. The **`To_Abstract_Ref`** helper function meets the substitutability requirements.

1.10.4.3 Interfaces Inheriting from Abstract Interfaces

Concrete (non-abstract) interfaces may also inherit from multiple abstract interfaces. The Ada mapping of inheritance from abstract interfaces is not significantly different from inheritance of non-abstract interfaces. Currently, Ada's tagged type inheritance is used for the first-named parent interface, and the operations of the remaining interfaces are "copied down." In the interface implementation package, the inheritance of the first-named parent allows implementation inheritance from that parent. Since abstract interfaces never have an implementation, implementation inheritance is never an issue. Therefore, the rules for inheritance are modified so that tagged-type inheritance is used for the first-named non-abstract interface in the list of inherited interfaces. The operations and attributes of all abstract interfaces are "copied down."

For example, the interface, **`Extended`**, in the above IDL is mapped to:

```
-- Ada - example-extended.ads
with Example.Marker;
package Example.Extended is

    type Ref is new Example.Marker.Ref with null record;

    procedure BaseOp(Self : Ref);

    function Method(Self : Ref; Arg : in CORBA.Long)
        return CORBA.Long;
    procedure Set_Assignable(Self: Ref; To: in CORBA.Long);
    function Get_Assignable(Self : Ref) return CORBA.Long;
    function Get_Nonassignable(Self : Ref)
        return CORBA.Long;

end Example.Extended;
```

1.10.4.4 *Base Types for Abstract Interfaces*

In order to support abstract interfaces, the following package is introduced:

```
-- Ada
package CORBA.AbstractBase is

    type Ref is new Ada.Finalization.Controlled with
        record
            Ptr : CORBA.Impl.Object_Ptr;
            ...
        end record;

    procedure Initialize (The_Ref : in out Ref);
    procedure Adjust     (The_Ref : in out Ref);
    procedure Finalize   (The_Ref : in out Ref);

    procedure Unref      (The_Ref : in out Ref)
                           renames Finalize;

    function Is_Nil(Self : in Ref) return Boolean;

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

    procedure Duplicate(Self : in out Ref) renames Adjust;

    procedure Release(Self : in out Ref);

    function Object_of(Self : Ref) return Object_Ptr;

end CORBA.AbstractBase;
```

Conforming implementations must support sharing semantics on assignment (of the implementation referred to), and reference counting for the implementations referred to.

In order to support the widening of both value types and interface types to abstract interfaces, the definition of **CORBA.Object.Ref** is modified so that it derives from **CORBA.AbstractBase.Ref**. Note that **CORBA.Value.Base** has already been specified as deriving from **CORBA.AbstractBase.Ref**.

Implementations will also require an (unspecified) common ancestor type for both **PortableServer.Servant** and **CORBA.Value.Impl_Base**.

1.10.5 Other CORBA/IIOP Specification Changes

1.10.5.1 Escaped Identifiers

The lexical mapping rules were adjusted by a previous Revision Task Force to account for escaped identifiers.

1.10.5.2 Additional Pseudo-Operations

There are a number of new type and operation definitions in the CORBA pseudo-objects. The following changes have been incorporated into this revision:

- in package **CORBA: RequestSeq**
- in package **CORBA.ORB**:
  - **send_multiple_requests_oneway**
  - **send_multiple_requests_deferred**
  - **poll_next_response**
  - **get_next_response**
  - **create_value_tc**
  - **create_value_box_tc**
  - **create_recursive_tc**
  - **create_abstract_interface_tc**
- dynamic Any interfaces are mapped according to the standard rules
- additional Interface Repository definitions are mapped according to the standard rules

1.10.6 Delegating Servants

The current mapping for servants is inheritance-based; object implementations are required to inherit from a common base class, **PortableServer.Servant**. In some instances, in particular when there already exists a legacy implementation that the developer would like to wrap, this requirement can be intrusive. Also, the current mapping only supports implementation inheritance from the first-named parent. Ada supports other useful implementation strategies (for example, building-block

approaches to achieving the effects of multiple-inheritance, that are best implemented with a “clean slate”). Other language mappings support an alternative mapping for object implementations that is delegation-based. For example, the C++ mapping requires the generation of a “tie class” that delegates calls on the object implementation to an instance of a “tied” class.

For these reasons, this mapping revision codifies an additional mapping for object implementations that is delegation-based. The form of the additional mapping is a generic package that can be used to “wrap” any type with the proper syntax; that is, it supports subprograms with the proper signatures, and yields a CORBA servant type that can be registered with a POA.

For each interface, an additional “implementation delegation” package is to be generated. The package will be a child package of the interface package with name extension **`.Delegate`**. The implementation delegation package will be generic with the following formal parameters:

1. A limited private type that is the type to be wrapped.

2. For each mapped attribute accessor/setter and mapped operation from the interface and all of its ancestors (not including **`CORBA.Object`**), a generic formal subprogram parameter with the same signature as the subprograms mapped for the **`.Impl`** packages will be required. The formal subprogram parameter will have the “is box” form of default.

The generic package will define a new type derived from **`PortableServer.Servant_Base`**. Instances of this type may be registered with a POA to service remote and local requests. This type, **`Object`**, will be declared with unknown discriminants so that instances may not be declared without initialization. A class-wide access type, **`Object_Ptr`**, will also be declared. Finally, a **`Create`** function will be declared that yields an **`Object_Ptr`** given an access to an instance of the wrapped type.

For example, for the **`horse`** interface described later in this document, the following implementation delegation package will be generated:

```
with Feed;
with PortableServer;
with Animal;
generic
    type Wrapped is limited private;
    with procedure eat (Self    : access Wrapped;
                        bag     : in out Feed.Ref;
                        Returns :    out CORBA.Boolean) is <>;
    with function Get_alertness (Self : access Wrapped)
        return Animal.State is <>;
    with procedure Set_alertness (Self : access Wrapped;
        To : in Animal.State) is <>;
    with function Get_parent (Self : access Wrapped)
        return Ref'CLASS is <>;
    with procedure trot (Self : access Wrapped;
        distance : in CORBA.Short) is <>;
```

```
package Horse.Delegate is

    type Object(<>) is new PortableServer.Servant_Base
        with private;
    type Object_Ptr is access all Object'CLASS;
    function Create(From : access Wrapped) return Object_Ptr;

end Horse.Delegate;
```

1.10.7 Changes from CORBA Components Specification

The CORBA Components specification introduced a number of changes into IDL.

1.10.7.1 Local Interfaces

Local interface types were introduced by the CORBA Components submission. Essentially, they are a formalization in IDL of the practice of defining “locality constrained” interfaces (for example, for interfaces to ORB services that can only be used locally). However, they are now also available for use by application developers.

The syntax of local interfaces differs only in the inclusion of the **`local`** keyword.

The semantics of local interfaces differ from “normal” (now termed unconstrained) interfaces in several ways:

- Instances of local interfaces cannot be “marshalled.” This includes passing after widening to an unconstrained base interface, stuffing into an any, or passing into **`Object_to_String`**.
- Inheritance: there are restrictions in inheritance so that local types do not have to be narrowed to unconstrained interfaces.
- Many of the predefined operations on the **`Object`** base class do not make sense for local objects. These operations are modified to return exceptions when invoked on local interfaces.

Otherwise, the language mapping requirements require unconstrained interfaces and local interfaces to be mapped in very similar ways.

A standard mapping is also required for implementations of local interfaces. The Components specification defines a **`LocalObject`** type as the base class for implementations. It also ascribes semantics to this implementation class that are more properly semantics of operations on the references to the implementation.

The Ada mapping for local interfaces produces three packages: an interface package, an interface implementation package, and a helper package. These packages will be very similar to those mapped for unconstrained interfaces.

1.10.7.2 Interface Package Mapping

The Ada interface package mapping for local interfaces differs from the interface package mapping (previously referred to as the "client-side mapping") in only one way: the name of the reference type will be **`Local_Ref`** in order to reinforce the semantic difference.

`CORBA.Object.Ref` continues to be the ultimate ancestor type for all interface reference types. However, the specification of the semantics of the subprograms primitive to **`CORBA.Object.Ref`** is altered to add the required semantics when the implementation referred to is an implementation of a local interface.

1.10.7.3 Implementation Package Mapping

The proposed Ada implementation package mapping for local interfaces differs only in the inheritance requirements for the **`Object`** type declared in the package. Instead of ultimately deriving from **`PortableServer.Servant`**, the implementation type will directly or indirectly derive from the new **`CORBA.Local.Object`** type.

1.10.7.4 Helper Package Mapping

The helper package for local interfaces need only contain a **`To_Local_Ref`** function, that is responsible for narrowing and widening to the interface package's **`Local_Ref`** type from ancestor local or unconstrained interfaces or from descendant local interface types.

Since instances of local interfaces cannot be inserted into an **`any`**, the helper package for a local interface should not contain the **`To_Any`** and **`From_Any`** subprograms needed for an unconstrained interface.

1.10.7.5 LocalObject Mapping

The components submission specifies a new base native type for local implementations. It also ascribes semantics to that, but, in the Ada mapping at least, these semantics are better ascribed to the object references that refer to them. The Ada mapping for **`LocalObject`** is the type **`CORBA.Local.Object`** defined in the following package:

```
package CORBA.Local is

    type Object is abstract tagged limited private;

end CORBA.Local;
```

Implementations will also require an (unspecified) common ancestor type for **`CORBA.Local.Object`**, **`PortableServer.Servant`**, and **`CORBA.Value.Impl_Base`**.

1.10.7.6 Forward Declarations

The rules for mapping forward declarations of local interfaces are the same as those for mapping of unconstrained interfaces.

1.10.7.7 Import

Visibility rules determine whether an identifier is usable when interpreting a set of IDL. The most important visibility rules govern the visibility of definitions defined outside the current scope of compilation. Prior to the Components submission, IDL visibility rules had been based on the use of pre-processor directives to “include” definitions within files into the compilation of a file. The Ada mapping for include was fairly straightforward: an include directive was reflected in the generated Ada by a corresponding “with” statement.

The CORBA Components submission defines a new IDL “import” statement that allows visibility to be selected on a unit or type basis. The subject of the import can be a qualified name or a string containing the repository ID of an IDL name scope. The specification of import also governs other aspects of IDL processing (for example, the ability to “re-open” a module).

Unfortunately, the IDL visibility rules for import conflict with the Ada visibility rules for withing. For example, “importing” an inner name scope in IDL does not implicitly import the definitions in the enclosing name scopes, while “withing” a child package in Ada does implicitly “with” the parent packages. In addition, import allows use of the unqualified form of an identifier for an imported definition.

Because of these problems, the determination of “withs” needed for the mapped Ada code must be decoupled from the visibility directives for IDL. The Ada mapping now contains no explicit mapping for include directives or import statements. The software compiling IDL to Ada must observe the IDL visibility rules to determine the legality of the IDL being processed, and then must separately determine the “with statements” needed to make the generated Ada code compilable.

1.10.7.8 Repository Identity Declarations

The Components submission replaces the IDL pragmas that define components of the repository ID for a type definition with explicit statements. These two new statements use the new keywords **`typeID`** and **`typePrefix`**.

There is no need for an Ada mapping of these new statements; their only effect is on the generated TypeCode constants for use with type **`any`**. This is described in the appropriate section of the mapping.

1.10.7.9 Exception Clauses for Attributes

The Components submission introduces exception clauses for interface attribute definitions. These clauses are introduced by the new IDL keywords **`getRaises`** and **`setRaises`**, which define the user-defined exceptions that may be raised when querying or setting the value of an attribute.

Like the raises clause for operations, the Ada mapping requires no mapping of these clauses.

Lexical Mapping 2

Contents

This chapter contains the following sections.

2.1 Mapping of Identifiers

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

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

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

2.2 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.

2.2.1 Integer Literals

IDL supports decimal, octal, and hexadecimal integer literals:

- A decimal literal consists of a sequence of digits that does not begin with 0 (zero). Decimal literals are lexically equivalent to Ada literal values and shall be mapped “as is.”
- An octal literal consists of a leading ‘0’ followed by a sequence of octal digits (0 .. 7). Octal constants shall be lexically mapped by prepending “8#” and appending “#” to the IDL literal. The leading zero in the IDL literal may be deleted or kept.
- A hexadecimal literal consists of “0x” or “0X” followed by a sequence of hexadecimal digits (0 .. 9, [a|A] .. [f|F]). Hexadecimal literals shall be lexically mapped to Ada literals by deleting the leading “0x” or “0X,” prepending “16#” and appending “#.”

2.2.2 Floating-Point Literals

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

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

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

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

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

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

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

2.2.3 Fixed Point Literals

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

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

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

2.2.4 Character Literals

IDL character literals are single graphic characters or escape sequences enclosed by single quotes. The first form is lexically equivalent to an Ada character literal. Table 2-1 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 2-1 Lexical Equivalents for the Defined Escape Sequences

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

2.2.5 Wide Character Literals

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

2.2.6 String Literals

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

- If the string literal does not contain escape sequences (does not contain ‘\’), the IDL literal is lexically equivalent to a valid Ada literal.
- If the IDL literal contains escape sequences, the string must be partitioned into substrings. As each embedded escape sequence is encountered, three partitions must be formed:
  - one containing a substring with the contents of the string before the escape sequence,
  - one containing the escape sequence only, and
  - one containing the remainder of the string.

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

2.2.7 Wide String Literals

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

2.2.8 Enumeration Literals

Enumeration literals are specified by IDL identifiers. Mapping rules for enumeration literals are the same as for identifiers (see Section 2.1, “Mapping of Identifiers,” on page 2-1).

2.3 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.

2.3.1 Mapping of Operators

Table 2-2 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 2-2 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	√	√		

Table 2-2 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
				-(value - 1)			√	

Mapping of IDL Types 3

Contents

This chapter contains the following sections.

Note that the following IDL semantics (from the *CORBA/IIOP Specification*) 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.

3.1 Mapping of Names

3.1.1 Identifiers

The lexical mapping of IDL identifiers is specified in Section 2.1, “Mapping of Identifiers,” on page 2-1. All identifiers in the Ada interfaces generated from IDL shall be mapped from the corresponding IDL identifiers.

3.1.2 Scoped Names

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

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

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

3.2 Mapping for Basic Types

Several basic numeric types are defined in IDL. These types shall be mapped to Ada (sub)types. The following Ada types shall be defined in the package “**`CORBA`**” with correspondence to IDL types, as shown in Table 3-1.

Table 3-1 Ada Types with Correspondence to IDL Types

Ada Type	IDL Type	Required Range and Representation
CORBA.Short	short	integer, range -(2**15) .. (2**15 - 1)
CORBA.Long	long	integer, range -(2**31) .. (2**31 - 1)
CORBA.Long_Long	long long	integer, range -(2**63) .. (2**63 - 1)
CORBA.Unsigned_Short	unsigned short	integer, range 0 .. (2**16 - 1)
CORBA.Unsigned_Long	unsigned long	integer, range 0 .. (2**32 - 1)

Table 3-1 Ada Types with Correspondence to IDL Types

Ada Type	IDL Type	Required Range and Representation
CORBA. Unsigned_Long_Long	unsigned long long	integer, range 0 .. (2**64 - 1)
CORBA.Float	float	floating point, ANSI/IEEE 754-1985 single precision
CORBA.Double	double	floating point, ANSI/IEEE 754-1985 double precision
CORBA.Long_Double	long double	floating point, ANSI/IEEE 754-1985 double extended precision
CORBA.Char	char	8 bit ISO Latin-1 (8859.1) character set
CORBA.Wchar	wchar	multi-byte character of negotiated character set
CORBA.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 3-2, should be used.

Table 3-2 Declarations

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

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

3.3 *Mapping for Fixed Type*

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

For example, the following IDL definition:

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

will map to:

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

3.4 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;
```

3.5 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);
```

3.6 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;
```

3.7 Mapping for Union Types

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

- **Discrete_choice_list**: For **case_labels** specified by "case" followed by a **const_exp**, the **const_exp** defines a **discrete_choice**. For the "default" **case_label**, the **discrete_choice** is "others." If more than one **case_label** is associated with a case, they shall be "or"ed together.
- Variant **component_list**: The **component_list** of each variant shall contain one component formed from the **element_spec** using the mapping in Section 3.6, "Mapping for Structure Types," on page 3-4 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;
```

3.8 Mapping for Sequence Types

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

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

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

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

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

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

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

The following sequence types in DrawingKit:

IDL File: drawing.idl

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

map to generic package instantiations, as follows:

```
package Fresco is
end Fresco;

with CORBA.Sequences;
with CORBA.Object;

package Fresco.DrawingKit is

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

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

end Fresco.DrawingKit;
```

Note that for the purposes of other rules, the “type mapped from” a sequence declaration is the “.Sequence” type of the instantiated package. This is relevant to the rules for Typedefs (Section 3.13, “Mapping for Typedefs,” on page 3-11) and for other template types. Thus, in the previous example, the instantiated “**`.Sequence`**” type is followed by a type derivation. Also, the following declaration:

```
typedef sequence<sequence<octet>> Ragged8;
```

will map to

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

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

type Ragged8 is new IDL_SEQUENCE_SEQUENCE_octet.Sequence
```

3.9 Mapping for String Types

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

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

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

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

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

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

For example, the IDL declaration:

```
typedef string Name;
```

maps to

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

while the following declaration:

```
typedef string<512> Title;
```

may map to

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

at the library level, and

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

in the corresponding interface package.

3.10 Mapping for Wide String Types

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

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

```
function To_CORBA_Wide_String
   (Source : Standard.Wide_String)
   return CORBA.Wide_String;
function To_Standard_Wide_String
   (Source : CORBA.Wide_String)
   return Standard.Wide_String;
```

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

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

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

For example, the IDL declaration:

typedef wstring WName;

maps to

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

while the following declaration:

typedef wstring<512> WTitle;

may map to

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

at the library level, and

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

in the corresponding interface package.

3.11 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 Section 3.6, “Mapping for Structure Types,” on page 3-4, “Mapping for Union Types” on page 3-5, and Section 3.12, “Mapping for Constants,” on page 3-10 for more information.

3.12 Mapping for Constants

An IDL constant shall map directly to an Ada constant. The Ada constant name shall be mapped from the identifier in the IDL declaration. The type of the Ada constant shall be mapped from the IDL **const_type** as specified elsewhere in this section. The value of the Ada constant shall be mapped from the IDL constant expression as specified in Section 2.3, “Mapping of Constant Expressions,” on page 2-4. 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;
```

3.13 Mapping for Typedefs

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

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

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

For example, the following IDL typedefs:

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

will be mapped to

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

3.14 Mapping for TypeCodes

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

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

The Ada mapping of TypeCode is provided by the pseudo-object **`CORBA.TypeCode.Object`** type declared in the **`CORBA.TypeCode`** package nested within the **`CORBA`** package (see Section 5.5.5, "TypeCode," on page 5-19). Its implementation is left unspecified. The primitive operations of TypeCode are mapped from the pseudo-IDL contained in the *CORBA/IIOP 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/IIOP 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.

3.15 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.

3.15.1 Handling Known Types

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

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

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

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

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

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

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

3.15.2 Handling Unknown Types

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

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

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

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

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

3.16 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.

3.16.1 Exception Identifier

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

For example, the following IDL exception declaration:

```
exception null_exception{};
```

will map to the following Ada exception declaration:

```
Null_Exception: exception;
```

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

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

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

3.16.2 Exception Members

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

The following declarations shall be contained in package **CORBA**:

```
type IDL_Exception_Members is abstract tagged null record;

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

3.16.2.1 Standard Exceptions

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

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

The following declarations shall exist in package **CORBA**:

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

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

For example, the IDL standard exception declaration below:

```
exception UNKNOWN ex_body;
```

maps to the following:

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

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

3.16.2.2 *Application-Specific Exceptions*

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

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

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

Note – The use of the strings associated with **Exception_Message** and **Exception_Information** in the language-defined package **Ada.Exceptions** may be used by the implementer 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);
```

3.16.2.3 *Example Use*

The following interface definition:

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

maps to the following in Ada:

```
package Stack is

...

   type Element is new CORBA.Long;

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

```
            Exception_Occurrence;
               To:       out Overflow_Members;

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

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

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

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

    ...

exception

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

    ...

end Use_stack;
```

Mapping of IDL Units 4

This chapter discusses the mapping of the following IDL unit constructs to the Ada programming language:

- Files
- Modules
- Interfaces, include unconstrained, abstract, and local interfaces
- Value types

Contents

This chapter contains the following sections.

4.1 Name Visibility

The rules for visibility for IDL and Ada are different, and have been complicated by the introduction of the "import statement." Because of the inherent differences in the visibility models, there is no transformational mapping from the IDL constructs controlling visibility to the Ada constructs controlling visibility.

The requirements for mapping IDL visibility to the Ada language are:

1. The visibility of an IDL identifier within an IDL name scope shall be determined according to the visibility rules in the *CORBA/IIOP Specification*.
2. The Ada generated from the mapping of IDL shall contain sufficient "with statements" consistent with the visibility rules of Ada to allow the resulting code to compile.

4.1.1 File Inclusion

The *CORBA/IIOP Specification* states that

> "Text in files included with a **#include** directive is treated as if it appeared in the including file."

The primary use of the preprocessor facility is to make available definitions from other IDL specifications and avoid redundant IDL type declarations.

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.

4.1.2 Import Statement

The IDL **import** statements offer finer grained visibility control:

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

The form of the statement is:

import <scoped_name>;

The statement “imports;” that is, makes visible, all identifiers defined in the **scoped_name** name scope and all identifiers in nested scopes. Conforming compilers must implement the IDL visibility rules as specified in the *CORBA/IIOP Specification*.

The presence of an import statement shall result in Ada “with clauses” to the Ada library units that contain the mapped Ada constructs for the identifiers from the imported scopes that are referenced.

4.1.3 CORBA Subsystem

The Ada mapping relies on some predefined types, packages, and functions. In the *CORBA/IIOP 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.

Note – Only the referenced identifiers must be visible in the generated code. It is acceptable that all identifiers in the imported scopes are visible.

4.2 Mapping of IDL Files

4.2.1 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.

4.2.2 Other Pre-Processing

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

4.2.3 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, changing any embedded spaces to underscores (‘_’), and appending the string **_IDL_File**. If all the IDL statements in a file are enclosed by a single module or interface definition, the generation of this “file package” is optional.

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

4.3 Mapping Modules

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

Top level modules; that is, those not enclosed by other modules shall be mapped to library packages. Modules nested within other modules shall be mapped to child packages of the corresponding package for the enclosing module. 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.

4.4 General Mapping for Units

IDL has three object oriented unit constructs:

1. *Abstract interfaces*: Abstract interfaces have references and operations, but no implementations. In some sense, abstract interfaces are “more primitive” than non-abstract interfaces and value types: non-abstract interfaces and valuetypes may be inherited from abstract interfaces, but not vice versa.

2. *Interfaces*: Interfaces have references, attributes, operations, and implementations. References to *unconstrained interfaces* can be passed across the network, while references to *local interfaces* can never leave their original execution scope.

3. *Valuetypes*: Valuetypes have references, state members, operations, and implementations. Valuetypes are “passed by value” over the network. Abstract valuetypes may not have state members or implementations.

The Ada mapping for these constructs reflects the commonality among these constructs in a common pattern for mapping these constructs, and in a hierarchy of base types (classes) that support their mapping.

4.4.1 Package Pattern for Mapping

The Ada mapping for each IDL unit construct follows a common pattern. This pattern is used in the mapping of interfaces, whether unconstrained, abstract, or local, and value types.

The mapping of a unit requires the generation of three packages:

1. ***The interface package***: This package will be a child of the package of the package mapped from the enclosing IDL name scope (a module or file) and have a final name component mapped from the IDL identifier for the unit. It will define a reference type and its primitive operations. The primitive operations are mapped from the attribute accessors, operations, or state members of the IDL unit.

2. ***The interface implementation package***: This package will be a child package of the interface package with a final name component of ".Impl". It will define an **Object** type and the signature of its primitive operations, which are to be implemented by the developer.

3. ***An interface helper package***: This package will be a child package of the interface package with extended name **.Helper**. It will contain:
   - The functions that support widening and narrowing of references.
   - The **TypeCode** constants and functions supporting conversion to and from type **any**.

Certain packages may not be needed for some constructs. In particular, no implementation package is needed for an abstract interface or abstract valuetypes, since there can be no implementation for these.

4.4.2 Base Types

The Ada mapping defines the following types to support the mapping of IDL's object-oriented constructs:

- **CORBA.AbstractBase.Ref**: The base type for all abstract interface reference types, and also for **CORBA.Object.Ref** and **CORBA.Value.Base**. Thus, it is the base type for all references. Each instance contains a class-wide access to an instance of the **CORBA.Impl.Object** type. This type is the base type for all implementations. The primitive operations on this type provide the required reference counting and memory management semantics.
- **CORBA.Object.Ref**: The base type for all reference types mapped from IDL interfaces, whether unconstrained or local. The access component in each instance of this type will access an instance derived from the type **PortableServer.Servant_Base**.
- **CORBA.Value.Base**: The base type for all references mapped from IDL valuetypes. The access component in each instance of this type will access an instance derived from the type **CORBA.Value.Impl_Base**.
- **CORBA.Impl.Object**: The abstract base type for all implementations.
- **PortableServer.Servant_Base**: The base type for all servants inherits from **CORBA.Impl.Object**.
- **CORBA.Value.Impl_Base**: The base type for all valuetype implementations inherits from **CORBA.Impl.Object**.

The relationship among these types is illustrated in the following diagram:

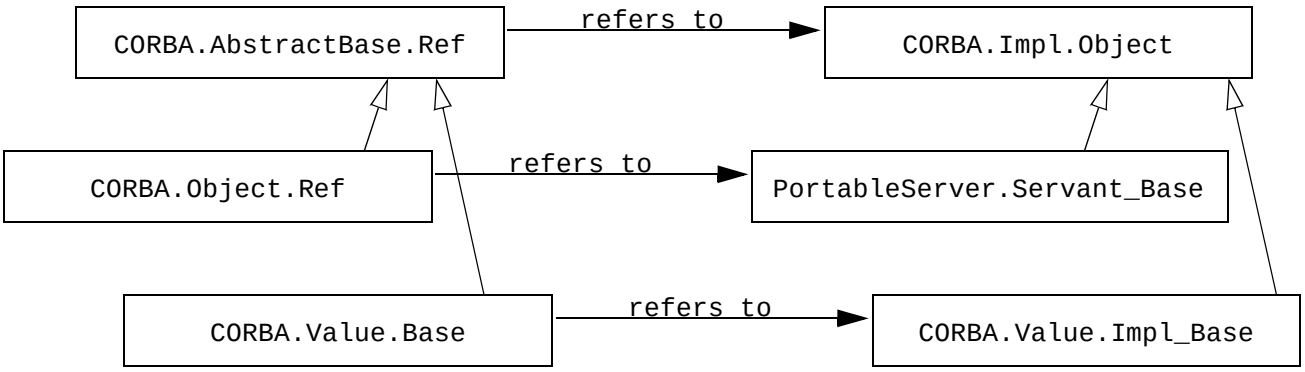

Figure 4-1 Base Types Relationship Diagram

Further details on these base types can be found in the “Mapping the CORBA Module” chapter.

4.5 Interface Package Mapping

Each IDL interface or valuetype 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 “reference” type, used to represent object references for the mapped interface or valuetype, as specified in Section 4.5.1, “Reference Types,” on page 4-7.

The declarations of constants, exceptions, and types scoped within interfaces or valuetype shall be mapped to declarations with the mapped Ada package, as specified in the “Mapping of IDL Types” chapter.

The declaration of attributes, operations, public state members, and initializers within each interface or valuetype shall be mapped to primitive subprograms of the reference type, according to the mapping specified in the following sections.

4.5.1 Reference Types

The reference type defined in the interface packages shall have a name as specified in Table 4-1. Each reference type shall be a controlled tagged type that will release automatically the memory of what it refers to, when it is deallocated, assigned a new object reference, or passes out of scope. A reference type is a private type; that is, its implementation is not visible to clients.

Table 4-1 Reference Type Names and Ancestor Types

IDL Construct	Name of Reference Type	Ancestor Type
abstract interface	`Abstract_Ref`	`CORBA.AbstractBase.Ref`
unconstrained interface	`Ref`	`CORBA.Object.Ref`
local interface	`Local_Ref`	
stateful valuetype	`Value_Ref`	`CORBA.Value.Base`
abstract valuetype	`Abstract_Value_Ref`	

4.5.2 Reference Type Inheritance

The reference type associated with a derived interface or valuetype will inherit all of the primitive subprograms of all of its parents as follows:

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

2. The attributes accessors, operations, and state member accessors of other parents (abstract or non-abstract) and (for valuetypes) supported interfaces will be generated by the IDL compiler. The signature of the generated operation will be mapped as for the parent interface, but the controlling parameter shall be of the child reference type.

If the interface or valuetype has an inheritance specification and the inheritance specification refers to a non-abstract parent of the same IDL construct (interface for interfaces or valuetype for valuetypes), the reference type shall be derived from the first-named non-abstract parent. If there is no inheritance specification or all parents named in the inheritance specification are abstract, the reference type shall be derived from the type named in the "Ancestor Type" column of Table 4-1.

Each attribute, operation, and public state member in each parent other than the parent derived from, from each supported interface, and from each of their parents transitively, shall be mapped to primitive subprograms of the reference type, according to the mapping specified in the following sections.

4.5.3 Mapping for Attributes and Public State Members

Attributes definitions may be contained in IDL definitions of abstract, unconstrained or local interfaces. Public state members may be contained in IDL definitions of stateful valuetypes. Note that private state members of valuetypes are not mapped into the interface package.

Read-only attributes shall be mapped to an Ada function with name formed by prepending **Get_** to the mapped attribute name. Read-write attributes and public state members 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 shall have a controlling parameter of the mapped reference type and name **Self**, and a parameter with name **To**. The type of the **To** parameter shall be mapped from the attribute or state member type, except for an attribute or state member of the enclosing unit type, which shall be mapped to the class of the reference type (for example, as **Ref'CLASS** for an interface). The **Get** function shall have a controlling parameter only, of the reference type and name **Self**. The **Get** function returns the type mapped from the attribute or state member type, except for an attribute or state member of the enclosing unit type, which shall be mapped to the class of the reference type (for example, as **Value_Ref'CLASS** for a valuetype).

Examples of mapped attributes may be found in Section 4.5.8, "Interface Mapping Examples," on page 4-10.

4.5.4 Mapping for Operations

Operation definitions may be contained in IDL definitions of abstract, unconstrained or interfaces, and in abstract or stateful valuetypes.

The IDL operations are mapped as primitive operations of the reference type. 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)**.

Operations shall map to an Ada subprogram with name mapped from the operation identifier. The first argument to operation subprograms will "refer" to the "subject" of the operation. The first argument to the mapped operation shall be an **in** mode argument with the name **Self** and shall be of the mapped reference type.

IDL 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 operations that have out-mode parameters, or void operations) operations shall be mapped to Ada procedures. The non-void result, if any, is returned via an added argument with name **Returns**. This result argument shall follow all other parameter arguments.

Each specified parameter in the operation declaration and the result type shall be mapped to an argument of the mapped subprogram. The argument names shall be mapped from the parameter identifier in the IDL. The argument mode shall be preserved from the IDL. The argument or return type shall be mapped from the IDL

type, except in the case of an argument or return type that is of the enclosing IDL unit type. Arguments or result types of the enclosing IDL unit type shall be mapped to the class of the mapped reference type (for example, to **Ref'CLASS** for unconstrained interfaces). This is necessary to prevent multiple controlling parameters (it removes the ambiguity as to which parameter controls dispatching.)

If an operation in an IDL specification has a context specification, then an additional argument with name **In_Context** of **in** mode and of type **CORBA.Context.Object** (see Section 5.5.4, "Context," on page 5-19) shall be added after all IDL specified arguments and before the **Returns** argument, if any. The **In_Context** argument shall have a default value of **CORBA.ORB.Get_Default_Context** (see Section 5.5.6, "ORB," on page 5-20).

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

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

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

Examples of mapped operations may be found in Section 4.5.8, "Interface Mapping Examples," on page 4-10.

4.5.5 Mapping for Valuetype Initializers

Definitions of initializers may be found in IDL for stateful valuetypes. Initializers are a portable means of defining the initial state of a value type.

An initializer definition shall be mapped to an Ada function that returns the mapped value reference type. (Thus they will be primitive on the value reference type.) The Ada function name shall be mapped from the IDL identifier for the name. Each parameter of the initializer shall be mapped to an argument of the function. Each argument shall be **in** mode, and shall have a name mapped from the parameter name in the initializer and shall of type mapped from the parameter type in the initializer.

An examples of a mapped initializer may be found in Section 4.5.9, "Valuetype Mapping Example," on page 4-12.

4.5.6 Argument Passing Considerations

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

4.5.7 Type Object

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

Type Object is a full (non-pseudo) object type. However, because it is the pre-defined root type for the Object class, its implementation does not conform to the mapping rules for interfaces and its implementation is left unspecified. See Section 5.2.2, “Object,” on page 5-3 for more information.

4.5.8 Interface Mapping Examples

The following IDL specification:

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

is mapped to these Ada interface 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;
end Feed;
```

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

   procedure Set_Alertness
      (Self : in Ref;
      To   : in State);
   function Get_Alertness
      (Self : in Ref) return State;
   function Get_Parent(Self : in Ref) return Ref'CLASS;
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);
end Horse;
```

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);

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);

end Vehicle;

with CORBA;
with Vehicle, Asset;
package Tank is

   type Ref is new Vehicle.Ref with null record;

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

end Tank;
```

4.5.9 Valuetype Mapping Example

The following IDL interface and valuetype specification:

```
// IDL
module ExampleB {

  interface Printer{
    typedef sequence<unsigned long> ULongSeq;
    void print(in ULongSeq data);
  };

  valuetype WeightedBinaryTree supports Printer {
    public long weight;
    private WeightedBinaryTree left;
    private WeightedBinaryTree right;
    factory createWBT(in long w);
    ULongSeq preOrder();
    ULongSeq postOrder();
  };
```

```
};
```

is mapped to these Ada interface packages:

```
-- Ada - exampleb-printer.ads
with CORBA.Object;
with CORBA.Unsigned_Long_Unbounded;
package ExampleB.Printer is

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

    type ULongSeq is ...

    procedure Print(Self: Ref; Data: in ULongSeq);

end ExampleB.Printer;

-- Ada - exampleb-weightedbinarytree.ads
with CORBA.Value;
with CORBA.Value;
with ExampleB.Printer;
package ExampleB.WeightedBinaryTree is

    type Value_Ref is new CORBA.Value.Base with null record;
    Null_Value : constant Value_Ref;

    function  Get_weight(Self : Value_Ref) return CORBA.Long;
    procedure Set_weight(Self : Value_Ref; To : CORBA.Long);

    function createWBT(w : in CORBA.Long) return Value_Ref;

    function preOrder (Self: Value_Ref)
        return ExampleB.Printer.ULongSeq;
    function postOrder (Self: Value_Ref)
        return ExampleB.Printer.ULongSeq;

    procedure print
        (Self :    Value_Ref;
         data : in Printer.ULongSeq);

end ExampleB.WeightedBinaryTree;
```

4.6 Helper Package Mapping

Each mapped interface package for an IDL interface or valuetype shall have a “helper package.” The helper package shall be a child package of the interface package with last name component **.Helper**. Each helper package shall define:

- A widening and narrowing subprogram that results in the mapped reference type defined in the interface package.

- A **TypeCode** constant and type **any** conversion functions for the mapped reference type and for all types defined in the interface package.

In addition, helper packages for stateful value that support unconstrained interfaces must provide additional supporting types and subprograms.

4.6.1 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. Widening is also directly supported by Ada for stateful valuetype references.

For example, 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 is defined for all mapped unit constructs. 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);
```

4.6.2 Narrowing Object References

Often it is necessary to convert a 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 shall provide a primitive subprogram in each helper package to perform and check the narrowing operation. Unlike widening, narrowing cannot be accomplished via normal Ada language mechanisms.

The provided function shall have a name formed by prepending **To_** to the name of the reference type. The function shall have one parameter with name **The_Ref** and type that is the class of the reference type, and shall have a return type of the reference type. For example, 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 or valuetype, regardless of whether the ancestor was defined through single or multiple inheritance. If **The_Ref** cannot be narrowed to the desired interface, this function shall raise CORBA.Bad_Param.

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

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

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

4.6.3 Type Any support

Each helper package, except those for local interfaces, shall define a **TypeCode** constant and conversion functions for the interface or valuetype with value as defined in the *CORBA/IIOP Specification*, and two conversion functions, **To_Any** and **From_Any**. These conversion functions shall insert and extract an instance of the reference type into/from an instance of type **any**. Each helper package shall define a **TypeCode** constant, **To_Any**, and **From_Any** for each type defined in the IDL unit mapped. The **TypeCode** constants shall have values as defined in the *CORBA/IIOP Specification*. The functions shall be defined according to the rules in Section 3.15, "Mapping for Any Type," on page 3-12. Implementations are permitted to suppress the generation of **TypeCodes** and type **any** conversion functions unless specifically required (for example, through a command line switch to the IDL compiler).

4.6.4 Valuetypes Supporting Interfaces

Valuetypes may "support" interfaces, a relationship that is similar to, but not exactly the same as, the interface inheritance (subtyping) relationship specified as the semantics for "inheritance" between interfaces. Abstract values may support multiple interfaces, while stateful values may only support one.

An instance of a value supporting an interface may be passed as a parameter of an operation that has a formal type of the interface (substitutability). The instance being passed must have previously been activated with a POA (an instance of a value type that supports an interface can be "widened" to that interface).

The value helper package for each valuetype that supports an interface shall, for each supported interface:

- Define a **Servant** and a **Servant_Ref** type. These types will "wrap" the **Value_Impl.Object** type defined in the implementation package for the valuetype. These types shall be defined as follows:

```
type Servant
   (Value: access
      mapped_implementation_object_type'CLASS) is
   new PortableServer.Servant with null record;
type Servant_Ref is access all Servant'CLASS;
```

- Define a **To_Servant** function that returns a **Servant_Ref** instance for an instance of the value's **Value_Impl.Object** type. This function shall have the following signature:

```
function To_Servant
   (Self: access mapped_implementation_object_type'CLASS)
   return Servant_Ref;
```

Once a **Servant_Ref** has been obtained, it can be activated with a POA and a reference to the supported interface can be obtained. This reference can be widened, if necessary.

4.6.5 Examples

The helper package for the **Animal** package would contain:

```
package Animal.Helper is
   TC_Ref : constant CORBA.TypeCode.Object;
   function To_Any(Item : Ref) return CORBA.Any;
   function From_Any(Item : CORBA.Any) return Ref;
   TC_State : constant CORBA.TypeCode.Object;
   function To_Any(Item : State) return CORBA.Any;
   function From_Any(Item : CORBA.Any) return State;
end Animal;
```

The helper package for the **ExampleB.WeightedBinaryTree** interface package would contain:

```
-- Ada - example-weightedbinarytree-helper.ads
with ExampleB.WeightedBinaryTree.Value_Impl;
with PortableServer;
package ExampleB.WeightedBinaryTree.Helper is
    function To_Any (From : in Value_Ref) return CORBA.Any;
    function From_Any (From : in CORBA.Any) return Value_Ref;
    TC_WeightedBinaryTree : constant CORBA.TypeCode.Object;

    type Servant
       (Value: access
       ExampleB.WeightedBinaryTree.Value_Impl.Object'CLASS)
      is
        new PortableServer.Servant with null record;
```

```
        type Servant_Ref is access all Servant'CLASS;
        function To_Servant
            (Self : access
            ExampleB.WeightedBinaryTree.Value_Impl.Object'CLASS)
            return Servant_Ref;

end ExampleB.WeightedBinaryTree.Helper;
```

4.7 Implementation Package Mapping

An “implementation package” shall be defined for each non-abstract IDL interface or valuetype. The implementation package shall be a child package of the interface package, with final name component of **`.Impl`** for interfaces or **`.Value_Impl`** for valuetypes. This implementation package shall define an “implementation” type, as specified in Section 4.7.1, “Implementation types,” on page 4-17.

The declaration of attributes, operations, and state members within each interface or valuetype shall be mapped to components of the implementation type or primitive subprograms of the implementation type, according to the mapping specified in the following sections.

4.7.1 Implementation types

The specification of the implementation package shall contain the declaration of a tagged record type, **`Object`**. The object record is used to hold member data employed by the implementation of the interface or value type.

The implementation package must define a class-wide general access type for the object class. The implementation package shall contain the following definition:

```
type Object_Ptr is access all Object’CLASS;
```

4.7.2 Implementation type inheritance

The implementation type associated with a derived interface or valuetype will inherit all of the primitive subprograms of all of its parents as follows:

1. The data members and operation implementations of the first-named non-abstract parent interface or valuetype will be inherited through Ada’s tagged type inheritance from the parent’s mapped implementation type.

2. The mapped attributes, and state member of other parents and (for valuetypes) supported interfaces will be added to the implementation type by the IDL compiler.

3. The operations of other parents (abstract or non-abstract) and (for valuetypes) supported interfaces will be generated by the IDL compiler. The signature of the generated operation will be mapped as for the parent interface, but the controlling parameter shall be of the child implementation type.

If the interface or valuetype has an inheritance specification and the inheritance specification refers to a non-abstract parent of the same IDL construct (interface for interfaces or valuetype for valuetypes), the implementation type shall be derived from the implementation type of the first-named non-abstract parent. If there is no inheritance specification or all parents named in the inheritance specification are abstract, the implementation type shall be derived from the type named in the "Ancestor Type" column of Table 4-2.

Table 4-2 Implementation Ancestor Types

IDL Construct	Ancestor Type
unconstrained interface	`PortableServer.Servant_Base`
local interface	`CORBA.Local.Object`
stateful valuetype	`CORBA.Value.Base`

Each state member in each parent other than the parent derived from, from each supported interface, and from each of their parents transitively, shall be mapped to component of the implementation type, according to the mapping specified in the following sections.

Each attribute, operation, and initializer in each parent other than the parent derived from, from each supported interface, and from each of their parents transitively, shall be mapped to primitive subprograms of the reference type, according to the mapping specified in the following sections.

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

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

4.7.3 Implementing Operations and Attributes

The parameters passed to an implementation subprogram parallel those passed to the corresponding subprogram in the interface package except that the subprograms will refer to type **Object**, not the reference type. Thus, all operation and attribute implementations will be primitive on the implementation type. This allows implementations to have a different inheritance hierarchy than that reflected in IDL. It allows inherited operations to be overridden by implementations (a facility that cannot be expressed in IDL). It also allows for alternate and delegating implementations that are not reflected in IDL.

A subprogram shall be generated for each mapped operation and attribute accessor generated in the interface package. The name and signature of each subprogram shall be the same as described in the mapping for the interface package in Section 4.5.3, “Mapping for Attributes and Public State Members,” on page 4-8 (for attributes only) and Section 4.5.4, “Mapping for Operations,” on page4-8, with the exception that type of the **`Self`** parameter shall be mapped as **`access Object`**, rather than the reference type declared in the interface package.

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

4.7.4 Implementing State Members

Each state member of a valuetype, public or private, must be a component of the implementation type. These state members must be marshalled when valuetypes are passed to remote servers in invocations.

The **`Object`** record for a stateful value type shall contain a component mapped from each state member. The component name shall be the mapped identifier of the state member, and the type shall be mapped from the IDL type of the state member.

4.7.5 Implementing Valuetype Initializers

An initializer definition shall be mapped to an Ada function that returns the **`Object_Ptr`** type. The Ada function name and signature shall be otherwise identical to the mapped function for the interface package.

4.7.6 Interface Implementation Example

The following IDL interface:

File cultivation.idl:

```
#include “barn.idl”

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

causes the IDL translator to generate the following implementation package specification:

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

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

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

4.7.7 Valuetype Implementation Example

The implementation package mapped from the **ExampleB.WeightedBinaryTree** valuetype is:

```
-- Ada - exampleb-weightedbinarytree-value_impl.ads
package ExampleB.WeightedBinaryTree.Value_Impl is

    type Object is new CORBA.Value.Impl_Base with record
        weight : CORBA.Long;
        left   : Value_Ref;
        right  : Value_Ref;
    end record;
    type Object_Ptr is access all Object'CLASS;

    function createWBT(w : in CORBA.Long) return Object_Ptr;

    function preOrder (Self: access Object)
        return ExampleB.Printer.ULongSeq;
    function postOrder (Self: access Object)
        return ExampleB.Printer.ULongSeq;

    procedure print
        (Self : access Object;
         Data : in     ExampleB.Printer.ULongSeq);
```

```
end ExampleB.WeightedBinaryTree.Value_Impl;
```

4.8 Delegating Servants

The mapping presented above for servants is inheritance-based; object implementations are required to inherit from a common base class, **`PortableServer.Servant`**. In some instances, in particular when there already exists a legacy implementation that the developer would like to wrap, this requirement can be intrusive. Also, this mapping only supports implementation inheritance from the first-named parent. Ada supports other useful implementation strategies (for example, building-block approaches to achieving the effects of multiple-inheritance) that are best implemented with a "clean slate."

For these reasons, an additional mapping for object implementations that is delegation-based is defined.

The form of the additional mapping is a generic package that can be used to "wrap" any type with the proper syntax; that is, it supports subprograms with the proper signatures, and yields a CORBA servant type that can be registered with a POA.

Implementation may suppress the generation of the package associated with this alternate mapping unless it is requested by the user (for example, through a command line compiler switch). When requested to generate this alternate mapping, the generation of the implementation package is optional.

When requested, an additional "implementation delegation" package shall be generated for each unconstrained and local interface. The implementation delegation package shall be a child package of the interface package with name extension **`.Delegate`**. The implementation delegation package shall be generic with the following formal parameters:

1. A limited private type with formal name **`Wrapped`**. This is the user type to be wrapped.

2. For each mapped attribute accessor/setter and mapped operation from the interface and all of its ancestors (not including **`CORBA.Object`**), a generic formal subprogram parameter with the same signature as the subprograms mapped for the **`.Impl`** packages shall be defined. The formal subprogram parameter will have the "is box" form of default.

The generic package shall define a new type with name **`Object`** derived (directly) from **`PortableServer.Servant_Base`**. Instances of this type may be registered with a POA to service remote and local requests. This type, **`Object`**, shall be declared with unknown discriminants. A class-wide access type, **`Object_Ptr`**, shall also be declared. A **`Create`** function shall be declared that yields an **`Object_Ptr`** from a general **`access`** to an instance of the **`Wrapped`** type.

For example, for the **`horse`** interface described in the current mapping document, the following implementation delegation package will be generated:

```
with Feed;
with PortableServer;
with Animal;
generic
    type Wrapped is limited private;
    with procedure eat (Self    : access Wrapped;
                        bag     : in out Feed.Ref;
                        Returns :    out CORBA.Boolean) is <>;
    with function Get_alertness (Self : access Wrapped)
        return Animal.State is <>;
    with procedure Set_alertness (Self : access Wrapped;
        To : in Animal.State) is <>;
    with function Get_parent (Self : access Wrapped)
        return Ref'CLASS is <>;
    with procedure trot (Self : access Wrapped;
        distance : in CORBA.Short) is <>;
package Horse.Delegate is

    type Object(<>) is new PortableServer.Servant_Base
        with private;
    type Object_Ptr is access all Object'CLASS;
    function Create(From : access Wrapped) return Object_Ptr;

end Horse.Delegate;
```

4.9 Mapping Forward Declarations

In IDL, a forward declaration defines the name of an interface or valuetype without defining it. This allows definitions of interfaces and valuetypes 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.

4.9.1 Forward Definition Packages

Conforming implementations shall provide two generic packages, **`CORBA.Forward`** and **`CORBA.Value_Forward`**, with the following specifications that will be used in the mapping of forward declarations.

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

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

```
            return Ref_Type
         renames From_Forward;
         function To_Forward (The_Ref : in Ref_Type)
            return Ref;
      end Convert;

   end CORBA.Forward;

   generic
   package CORBA.Value.Forward is

      type Value_Ref is new CORBA.Value.Base with null record;

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

   end CORBA.Value.Forward;
```

4.9.2 Mapping Rules

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

Within the full declaration of the forward declared interface or valuetype, the nested **`Convert`** package shall be instantiated with the actual **`Ref`** or **`Value_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`** or **`Value_Ref`** to the forward **`Ref`** or **`Value_Ref`** and vice versa. Clients holding an instance of a valid reference for an interface or valuetype may have to convert those references to the corresponding forward references for references mapped before the full declaration.

4.9.3 Example

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

File chicken.idl:

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

File egg.idl:

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

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

```
package Chicken_IDL_FILE is

end Chicken_IDL_FILE;

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

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

with CORBA.Object;
with Chicken_Forward;
with Egg_Forward;

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

```
with CORBA.Object;
with Egg;
with Chicken_Forward;

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

4.10 Mapping Value Boxes

The *CORBA/IIOP Specification* states this about value boxes:

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

4.10.1 Value Box Package

Conforming implementations shall provide an implementation of the following generic package:

```
generic
    type Boxed is private;
    type Boxed_Access is access all Boxed;
package CORBA.Value.Box is

    type Box_Ref is new CORBA.Value.Base with private;

    function Is_Null(The_Ref : Box_Ref) return Boolean;
    function Create(With_Value : in Boxed) return Box_Ref;
    function "+"   (With_Value : in Boxed) return Box_Ref
        renames Create;

    function Contents(The_Boxed : in Box_Ref)
        return Boxed_Access;
    function     "-" (The_Boxed : in Box_Ref)
        return Boxed_Access renames Contents;

    procedure Release(The_Ref : in out Box_Ref);

end CORBA.Value.Box;
```

Implementations of the **`Box_Ref`** type shall support reference counting, "smart pointer" semantics for the boxed value.

4.10.2 Mapping of Value Boxes

Each IDL value box declaration shall be mapped by:

- The declaration of a general access to the type to be boxed.
- An instantiation of **`CORBA.Value.Box`** with the type to be boxed and the previously declared access type as actual parameters of the instantiation. The name of the instantiated package shall be formed by appending **`_Value_Box`** to the IDL identifier for the value box.
- A derivation of the **`Box_Ref`** type from the instantiation with name mapped from the identifier of the value box shall be defined. This has the effect of introducing the type and its operations into the original name scope.

4.10.3 Example

For example, the following IDL:

```
// IDL
module Example {

   valuetype LongSeq sequence<Long>;
   interface Bar {
      void doit(in LongSeq seq1);
   };
};
```

maps to:

```
-- Ada - example.ads
package Example is

    package IDL_SEQUENCE_Long is new
        CORBA.Sequences.Unbounded(...

    type IDL_SEQUENCE_Long_Access is
        access all IDL_SEQUENCE_Long.Sequence;
    package LongSeq_Value_Box is
        new CORBA.Value.Box(IDL_SEQUENCE_Long.Sequence,
                            IDL_SEQUENCE_Long_Access);
    type LongSeq is new LongSeq_Value_Box.Box_Ref;

end Example;
-- Ada - example-bar.ads
with CORBA.Object;
package Example.Bar is

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

    procedure Doit(Self : in Ref; seq1: in Example.LongSeq);
```

```
end Example.Bar;
```

4.11 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 that 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.

Mapping the CORBA Module 5

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.

Contents

This chapter contains the following sections.

5.1 Mapping Rules for Pseudo-Objects

The types representing CORBA pseudo-objects are not derived from **`CORBA.Object.Ref`**. Ada also supports “object semantics” better than some other OOPLs. This mapping allows the types associated with pseudo-objects to be named

Object and support copy semantics in assignment. The **Self** parameter will be of the **Object** type and **in out** mode, except when the operation is obviously a query-only function, in which case the **Object** parameter is **in** mode.

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

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

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

In general, the operations for CORBA pseudo-objects are mapped from the pseudo-IDL according to the rules specified in the preceding chapter for local interfaces.

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

5.2 Reference and Implementation Base Types

The base types for references and implementation of CORBA’s object-oriented constructs were introduced in Section 4.4.2, “Base Types,” on page 4-5. Their complete definition is presented here.

5.2.1 AbstractBase

This native type is the base type of all abstract interfaces:

> “Abstract interfaces implicitly inherit from **CORBA::AbstractBase**. This type is defined as native. It is the responsibility of each language mapping to specify the actual programming language type that is used for this type.”

AbstractBase is mapped to a child package of the CORBA package. It contains an opaque implementation-defined tagged type that becomes the ancestor type for all references to interfaces (abstract, unconstrained, or local), valuetypes (stateful and abstract), and value boxes. This package has the following definition:

```
package CORBA.AbstractBase is

    type Ref is new Ada.Finalization.Controlled with
        record
            Ptr       : CORBA.Impl.Object_Ptr;
            ...
        end record;

    procedure Initialize (The_Ref : in out Ref);
```

```
        procedure Adjust     (The_Ref : in out Ref);
        procedure Finalize   (The_Ref : in out Ref);

        procedure Unref      (The_Ref : in out Ref)
            renames Finalize;

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

        procedure Duplicate(Self : in out Ref) renames Adjust;

        procedure Release(Self : in out Ref);

        function Object_Of(Self : Ref)
            return CORBA.Impl.Object_Ptr;

end CORBA.AbstractBase;
```

CORBA defines three operations on any reference type: **`duplicate`**, **`release`**, and **`is_nil`**. Note that these operations are on the reference type, not the implementation type. The Ada mapping of these operations are defined to be primitive to the **`CORBA.AbstractBase.Object`** type. Thus they will be inherited by all reference types, whether for abstract interfaces, unconstrained interfaces, local interfaces, stateful valuetypes, or abstract valuetypes.

Conforming implementations shall provide reference counting semantics for references such that the memory for an implementation may be reclaimed after the last local reference to that implementation has been finalized. The provided implementation of these operations shall have the following semantics:

- **`Duplicate`** shall be a renaming of **`Adjust`**. In general, explicit use of **`Duplicate`** by developers is not needed.
- The **`Is_Nil`** operation returns **`TRUE`** if the object reference contains an empty reference.
- The **`Release`** procedure indicates that the caller will no longer access the reference so that associated resources may be deallocated. If the given object reference is nil, **`Release`** does nothing. After a call to **`Release`**, a call to **`Is_Nil`** on the same reference must return **`TRUE`**.

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

5.2.2 Object

`Object` is the root of the IDL interface reference hierarchy. While **`Object`** is a normal CORBA object (not a pseudo-object), its interface is described here because it references other pseudo-objects and its implementation will necessarily be different.

The package **CORBA.Object** provides the Ada interface and includes a **Ref** type that is the root for client-side interfaces. See Section 4.5, “Interface Package Mapping,” on page 4-6 for more information.

```
--IDL: interface Object {
package CORBA.Object is

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

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

    -- IDL: boolean is_nil();
    -- inherited from CORBA.AbstractBase.Ref

    -- IDL: Object duplicate();
    -- inherited from CORBA.AbstractBase.ref
    -- Note: not needed for assignment, just use :=

    -- IDL: void release();
    -- inherited from CORBA.AbstractBase.Ref)

    function Is_A (Self : in Ref;
                   Logical_Type_ID : in Standard.String)
        return Boolean;

    function Non_Existent (Self : Ref) return Boolean;

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

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

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

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

    function Get_Domain_Managers(Self : Ref)
        return CORBA.Domainmanager.DomainManagerList;
```

```
        procedure Set_Policy_Overrides
            (Self     : Ref;
             Policies : CORBA.Policy.PolicyList;
             Set_Add  : SetOverrideType);

    end CORBA.Object;
```

5.2.3 CORBA.Value.Base and CORBA.Value.Impl_Base

CORBA.Value.Base is the defined root for all references to valuetypes. It is a specialization of **CORBA.AbstractBase.Ref**. The **CORBA.Value** package also defines the **CORBA.Value.Impl_Base** type, which is the ancestor type for all value implementations. The **CORBA.Value** package has the following definition:

```
package CORBA.Value is

    type Base is abstract new CORBA.AbstractBase.Ref
        with null record;
    type Impl_Base is abstract tagged limited private;

end CORBA.Value;
```

5.2.4 CORBA.Impl.Object

CORBA.Impl.Object is the abstract ancestor type of all implementations, both of interfaces and valuetypes. It is defined in package **CORBA.Impl**:

```
package CORBA.Impl is

    type Object is abstract
      new Ada.Finalization.Limited_Controlled with
        record
            ... // must include reference count
        end record;

    type Object_Ptr is access all Object'CLASS;

    ...

end CORBA.Impl;
```

5.2.5 LocalObject

LocalObject is the ancestor type for all implementations of local interfaces. It is defined in package **CORBA.Local**:

```
package CORBA.Local is
```

```
        type Object is abstract tagged limited private;

end CORBA.Local;
```

5.2.6 PortableServer.Servant_Base

This type is the base type for all user implementations of interface implementations:

> "This specification defines a native type **PortableServer::Servant**. Values of the type **Servant** are programming-language-specific implementations of CORBA interfaces. Each language mapping must specify how **Servant** is mapped to the programming language data type that corresponds to an object implementation. The **Servant** type has the following characteristics and constraints."

The Ada mapping of this native type is very similar to the C++ mapping, and is specified as part of the **PortableServer** package. This package is the mapping of the **PortableServer** module:

```
package PortableServer is

    package POA_Forward is new CORBA.Forward;

    --I native Servant;
    type Servant_Base is abstract tagged limited private;
    type Servant is access all Servant_Base'CLASS;

    --I "values of type Servant support a language-specific
    --I programming interface that can be used by the ORB to
    --I obtain a default POA for that servant. This
    --I interface is used only to support implicit
    --I activation."
    function Get_Default_POA (For_Servant : in Servant_Base)
       return POA_Forward.Ref;

    --I "Values of type Servant provide default
    --I implementations of the standard object reference
    --I operations get_interface, is_a, and non_existent."
    function Get_Interface(For_Servant : Servant_Base)
         return CORBA.Object.Ref'CLASS;
    function Is_A         (For_Servant : Servant_Base;
                           Logical_Type_ID : Standard.String)
                           return Boolean;
    function Non_Existent (For_Servant : Servant_Base)
                           return Boolean;

    --I "Values of type Servant must be testable for identity
    function "="(Left, Right: in Servant_Base)
        return Boolean;
```

5.3 Mapping for Native Types

There are a number of types in the *CORBA/IIOP Specification* that are of **native** type. All language mappings must provide explicit mappings of native types.

5.3.1 AbstractBase

The mapping of **AbstractBase** has been discussed in Section 5.2.1, “AbstractBase,” on page 5-2.

5.3.2 ValueFactory

There is no need for an explicit **ValueFactory** type in the Ada mapping. There is no need for user visibility to this underlying type.

5.3.3 OpaqueValue

OpaqueValue is used in the definition of the Dynamic Invocation Interface. The Ada mapping has the following definition:

```
subtype OpaqueValue is System.Address;
```

5.3.4 PortableServer::Servant

The mapping of this native type has already been described in Section 5.2.6, “PortableServer.Servant_Base,” on page 5-6.

5.3.5 Cookie

The **Cookie** type allows the developer of **ServantLocator**s to pass arbitrary data between pairs of calls to Preinvoke and Postinvoke. It is defined as part of the **PortableServer.ServantLocator** as follows:

```
package PortableServer.ServantLocator is

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

    --I native Cookie;
    type Cookie_Base is -- ... implementation defined
                        -- must be tagged
    type Cookie is access all Cookie_Base'CLASS;

    ...
```

5.4 The CORBA package

The CORBA package contains the mapping of most of the definitions in the CORBA module that are not otherwise contained in interfaces or nested modules. The Ada mapping has moved some of the definitions into package where they are used and segmented some of the definitions into child packages. In particular, Interface Repository definitions have been moved to a child package, and so are not present here.

The specification of the **CORBA** package is:

```
--IDL: 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.

    type    Short            is new Interfaces.Integer_16;
    type    Long             is new Interfaces.Integer_32;
    type    Long_Long        is new Interfaces.Integer_64;
    type    Unsigned_Short   is new Interfaces.Unsigned_16;
    type    Unsigned_Long    is new Interfaces.Unsigned_32;
    type    Unsigned_Long_Long is new Interfaces.Unsigned_64
    type    Float            is new Interfaces.IEEE_Float_32;
    type    Double           is new Interfaces.IEEE_Float_64;
    type    Long_Double      is new Interfaces.IEEE_Extended_Float;

    subtype Char               is Standard.Character;
    subtype Wchar              is Standard.Wide_Character;
    type    Octet              is new Interfaces.Unsigned_8;

    subtype Boolean            is Standard.Boolean;

    type    String             is
       new Ada.Strings.Unbounded.Unbounded_String;
    function  To_CORBA_String (Source : Standard.String)
        return CORBA.String;
    function  To_Standard_String (Source : CORBA.String)
        return Standard.String;
    Null_String : constant String := To_CORBA_String ("");

    type    Wide_String is new
        Ada.Strings.Wide_Unbounded.Unbounded_Wide_String;
    function  To_CORBA_Wide_String
        (Source : Standard.Wide_String)
         return CORBA.Wide_String;
```

```
function  To_Standard_Wide_String
    (Source : CORBA.Wide_String)
     return Standard.Wide_String;
Null_Wide_String : constant Wide_String
    := To_CORBA_Wide_String ("");

--IDL: typedef string Identifier;
type Identifier is new CORBA.String;
Null_Identifier : constant Identifier
    := Identifier(Null_String);

--IDL: typedef string RepositoryId;
type RepositoryId is new CORBA.String;
Null_ID : constant RepositoryId
    := RepositoryId(Null_String);

--IDL: typedef string ScopedName;
type ScopedName is new CORBA.String;
Null_ScopedName : constant ScopedName
     := ScopedName(Null_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;

procedure Raise_Exception
    (Self : IDL_Exception_Members'CLASS);

-- CORBA 4.14 Standard Exceptions:
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 CORBA.IDL_Exception_Members with
    record
        Minor     : CORBA.Unsigned_Long;
        Completed : CORBA.Completion_Status;
    end record;
procedure Get_Members
    (From : in Ada.Exceptions.Exception_Occurrence;
     To   : out System_Exception_Members);

-- the unknown exception
UNKNOWN : exception;
type UNKNOWN_Members is new System_Exception_Members
    with null record;
```

```
-- an invalid parameter was passed
BAD_PARAM : exception;
type BAD_PARAM_Members is new System_Exception_Members
    with null record;

-- dynamic memory allocation failure
NO_MEMORY : exception;
type NO_MEMORY_Members is new System_Exception_Members
    with null record;

-- violated implementation limit
IMP_LIMIT : exception;
type IMP_LIMIT_Members is new System_Exception_Members
    with null record;

-- communication failure
COMM_FAILURE : exception;
type COMM_FAILURE_Members is
    new System_Exception_Members with null record;

-- invalid object reference
INV_OBJREF : exception;
type INV_OBJREF_Members is new System_Exception_Members
    with null record;

-- no permission for attempted op.
NO_PERMISSION : exception;
type NO_PERMISSION_Members is
    new System_Exception_Members with null record;

-- ORB internal error
INTERNAL : exception;
type INTERNAL_Members is new System_Exception_Members
    with null record;

-- error marshalling param/result
MARSHAL : exception;
type MARSHAL_Members is new System_Exception_Members
    with null record;

-- ORB initialization failure
INITIALIZATION_FAILURE : exception;
type INITIALIZATION_FAILURE_Members is
    new System_Exception_Members with null record;

-- operation implementation unavailable
NO_IMPLEMENT : exception;
type NO_IMPLEMENT_Members is
    new System_Exception_Members with null record;

-- bad typecode
BAD_TYPECODE : exception;
type BAD_TYPECODE_Members is
    new System_Exception_Members with null record;
```

```
-- invalid operation
BAD_OPERATION : exception;
type BAD_OPERATION_Members is
    new System_Exception_Members with null record;

-- insufficient resources for req.
NO_RESOURCES : exception;
type NO_RESOURCES_Members is
    new System_Exception_Members with null record;

-- response to request not yet available
NO_RESPONSE : exception;
type NO_RESPONSE_Members is
    new System_Exception_Members with null record;

-- persistent storage failure
PERSIST_STORE : exception;
type PERSIST_STORE_Members is
    new System_Exception_Members with null record;

-- routine invocations out of order
BAD_INV_ORDER : exception;
type BAD_INV_ORDER_Members is
    new System_Exception_Members with null record;

-- transient failure - reissue request
TRANSIENT : exception;
type TRANSIENT_Members is
    new System_Exception_Members with null record;

-- cannot free memory
FREE_MEM : exception;
type FREE_MEM_Members is
    new System_Exception_Members with null record;

-- invalid identifier syntax
INV_IDENT : exception;
type INV_IDENT_Members is
    new System_Exception_Members with null record;

-- invalid flag was specified
INV_FLAG : exception;
type INV_FLAG_Members is
    new System_Exception_Members with null record;

-- error accessing interface repository
INTF_REPOS : exception;
type INTF_REPOS_Members is
    new System_Exception_Members with null record;

-- error processing context object
BAD_CONTEXT : exception;
type BAD_CONTEXT_Members is
    new System_Exception_Members with null record;
```

```
-- failure detected by object adapte
OBJ_ADAPTER : exception;
type OBJ_ADAPTER_Members is
    new System_Exception_Members with null record;

-- data conversion error
DATA_CONVERSION : exception;
type DATA_CONVERSION_Members is
    new System_Exception_Members with null record;

-- object not exist error
OBJECT_NOT_EXIST : exception;
type OBJECT_NOT_EXIST_Members is
    new System_Exception_Members with null record;

-- required transaction context not present
TRANSACTION_REQUIRED : exception;
type TRANSACTION_REQUIRED_Members is
    new System_Exception_Members with null record;

-- transaction already rolled back or marked for
-- rollback, cannot proceed
TRANSACTION_ROLLEDBACK : exception;
type TRANSACTION_ROLLEDBACK_Members is
    new System_Exception_Memberswith null record;

-- invalid transaction context
INVALID_TRANSACTION : exception;
type INVALID_TRANSACTION_Members is
    new System_Exception_Members with null record;

-- invalid policy
INV_POLICY : exception;
type INV_POLICY_Members is
    new System_Exception_Members with null record;

-- invalid transaction context
CODESET_INCOMPATIBLE : exception;
type CODESET_INCOMPATIBLE_Members is
    new System_Exception_Members with null record;

-- TypeCodes

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

type ValueModifier is new Short;
VTM_NONE        : constant ValueModifier := 0;
```

```
VTM_CUSTOM      : constant ValueModifier := 1;
VTM_ABSTRACT    : constant ValueModifier := 2;
VTM_TRUNCATABLE : constant ValueModifier := 3;

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


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

-- IDL: interface TypeCode

package TypeCode is

    type Object is private;

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

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

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

    function Equivalent(Left, Right : in Object)
        return Boolean;
    function Get_Compact_Typecode(Self : in Object)
        return Object;

    function Kind (Self : in Object) return TCKind;

    --IDL: for tk_objref, tk_struct, tk_union, tk_enum,
    --IDL: tk_alias, tk_value, tk_value_box, tk_native,
    --IDL: tk_abstract_interface, and tk_except
    function Id (Self : in Object) return RepositoryId;

    --IDL: for tk_objref, tk_struct, tk_union, tk_enum,
    --IDL: tk_alias, tk_value, tk_value_box, tk_native,
    --IDL: tk_abstract_interface, and tk_except
    function Name (Self : in Object) return Identifier;
```

```
        --IDL: for tk_struct, tk_union, tk_enum, tk_value,
        --IDL: tk_value_box, and tk_except
        function Member_Count (Self : in Object)
            return Unsigned_Long;
        function Member_Name (Self : in Object;
            Index : in Unsigned_Long) return Identifier;

        --IDL: for tk_struct, tk_union, tk_value,
        --IDL: tk_value_box, and tk_except
        --IDL: TypeCode member_type (in unsigned long index)
        --IDL:   raises (BadKind, Bounds);
        function Member_Type (Self : in Object;
            Index : in Unsigned_Long) return Object;

        --IDL: for tk_union
        function Member_Label (Self : in Object;
            Index : in Unsigned_Long) return Any;
        function Discriminator_Type (Self : in Object)
            return Object;
        function Default_Index (Self : in Object)
            return Long;

        --IDL: for tk_string, tk_sequence, and tk_array
        function Length (Self : in Object)
            return Unsigned_Long;

        --IDL: for tk_sequence, tk_array, tk_value,
        --IDL: tk_value_box, and tk_alias
        function Content_Type (Self : in Object)
            return Object;

        --IDL: for tk_fixed
        function Fixed_Digits(Self : in Object)
            return Unsigned_Long;
        function Fixed_Scale(Self : in Object) return Short;

        --IDL: for tk_value
        function Member_Visibility(Self : in Object;
            Index : in Unsigned_Long) return Visibility;
        function Type_Modifier(Self : in Object)
            return ValueModifier;
        function Concrete_Base_Type(Self : in Object)
            return Object;

    end TypeCode;

    -- pre-defined TypeCode "constants"
    function TC_null               return TypeCode.Object;
    function TC_void               return TypeCode.Object;
    function TC_short              return TypeCode.Object;
    function TC_long               return TypeCode.Object;
    function TC_long_long          return TypeCode.Object;
    function TC_unsigned_short     return TypeCode.Object;
    function TC_unsigned_long      return TypeCode.Object;
    function TC_unsigned_long_long return TypeCode.Object;
```

```
function TC_float              return TypeCode.Object;
function TC_double             return TypeCode.Object;
function TC_long_double        return TypeCode.Object;
function TC_boolean            return TypeCode.Object;
function TC_char               return TypeCode.Object;
function TC_wchar              return TypeCode.Object;
function TC_cctet              return TypeCode.Object;
function TC_any                return TypeCode.Object;
function TC_TypeCode           return TypeCode.Object;
-- TC_Object defined in CORRBA.Object
function TC_string             return TypeCode.Object;
function TC_wide_string        return TypeCode.Object;


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

function To_Any (From : in Short)             return Any;
function To_Any (From : in Long)              return Any;
function To_Any (From : in Long_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 Unsigned_Long_Long)return Any;
function To_Any (From : in Float)             return Any;
function To_Any (From : in Double)            return Any;
function To_Any (From : in Long_Double)       return Any;
function To_Any (From : in Boolean)           return Any;
function To_Any (From : in Char)              return Any;
function To_Any (From : in WChar)             return Any;
function To_Any (From : in Octet)             return Any;
function To_Any (From : in Any)               return Any;
function To_Any (From : in TypeCode.Object)   return Any;
function To_Any (From : in CORBA.String)      return Any;
function To_Any (From : in CORBA.Wide_String) return Any;

function From_Any (From : in Any) return Short;
function From_Any (From : in Any) return Long;
function From_Any (From : in Any) return Long_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 Unsigned_Long_Long;
function From_Any (From : in Any) return Float;
function From_Any (From : in Any) return Double;
function From_Any (From : in Any) return Long_Double;
function From_Any (From : in Any) return Boolean;
function From_Any (From : in Any) return Char;
function From_Any (From : in Any) return WChar;
function From_Any (From : in Any) return Octet;
function From_Any (From : in Any) return Any;
function From_Any (From : in Any) return TypeCode.Object;
function From_Any (From : in Any) return CORBA.String;
function From_Any (From : in Any) return CORBA.Wide_String;
```

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

        type Flags is new CORBA.Unsigned_Long;
        ARG_IN     : constant Flags;
        ARG_OUT    : constant Flags;
        ARG_INOUT  : constant Flags;


        type NamedValue is record
            Name        : Identifier;
            Argument    : Any;
            Arg_Modes   : Flags;
        end record;

        OUT_LIST_MEMORY     : constant Flags;
        IN_COPY_VALUE       : constant Flags;
        INV_NO_RESPONSE     : constant Flags;
        INV_TERM_ON_ERR     : constant Flags;
        RESP_NO_WAIT        : constant Flags;
        DEPENDENT_LIST      : constant Flags;
        CTX_RESTRICT_SCOPE  : constant Flags;

        -- in support of Object interface
        type PolicyType is new Unsigned_Long;
        type SetOverrideType is (SET_OVERRIDE, ADD_OVERRIDE);

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

    end CORBA;
```

5.5 Other Pseudo-Objects

The *CORBA/IIOP Specification* defines a number of pseudo-interfaces that must be explicitly mapped by the language specification. These types and packages are presented here. Packages mapped from the **`PortableServer`** module and other local interfaces are not listed.

5.5.1 NamedValue

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

```
type Flags is new CORBA.Unsigned_Long;
ARG_IN: constant Flags;
ARG_OUT: constant Flags;
ARG_INOUT: constant Flags;
```

```
IN_COPY_VALUE: constant Flags;
type NamedValue is record
Name        : Identifier;
Argument    : Any;
Arg_Modes   : Flags;
end record;
```

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

```
package CORBA.NVList is

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

    procedure Add_Item
        (Self       :    Ref;
         Item_Name  : in Identifier;
         Item_Type  : in CORBA.TypeCode.Object;
         Value      : in System.Address;
         Len        : in Long;
         Item_Flags : in Flags);
    procedure Add_Item
        (Self       :    Ref;
         Item_Name  : in Identifier;
         Item       : in CORBA.Any;
         Item_Flags : in Flags);
    procedure Add_Item
        (Self       :    Ref;
         Item       : in NamedValue);

    -- free and free_memory not needed in Ada
    procedure Free(Self : Ref);
    procedure Free_Memory(Self : Ref);

    function Get_Count (Self : Ref) return CORBA.Long;

end CORBA.NVList;
--IDL: };
```

5.5.3 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 that an additional version of **Add_Arg** is provided that takes a **NamedValue**.

```
package CORBA.Request is

    type Object is private;

    procedure  Add_Arg
        (Self      : in out Object;
         Arg_Type  : in     CORBA.TypeCode.Object;
         Value     : in     System.Address;
         Len       : in     Long;
         Arg_Flags : in     Flags);
    procedure  Add_Arg
        (Self : in out Object;
         Arg  : in     NamedValue);

    procedure Invoke
                (Self         : in out Object;
                 Invoke_Flags : in     Flags := 0);

    procedure Delete(Self : in out Object);

    procedure Send
        (Self         : in out Object;
         Invoke_Flags : in     Flags  := 0);

    procedure Get_Response
        (Self           : in out Object;
         Response_Flags : in     Flags  := 0);

    function Poll_Response(Self : in Object) return Boolean;

end CORBA.Request;
```

5.5.4 Context

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

```
package CORBA.Context is

   type Ref is private;

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

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

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

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

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

   procedure Delete
      (Self      : in Ref;
       Del_Flags : in Flags);

end CORBA.Context;
```

5.5.5 TypeCode

A **TypeCode** represents IDL type information. It is intimately related to type **Any**. For this reason, package **TypeCode** that defines the **Object** type for **TypeCode** is a subpackage nested within the **CORBA** package. See Section 3.14, “Mapping for TypeCodes,” on page 3-12 for more information.

The **TypeCode** type is used by type **any** and type **any** is used by **TypeCode**. Because of this circularity, the **TypeCode** package is defined as a nested subpackage of the CORBA package. Its definition can be found in Section 5.4, “The CORBA package,” on page 5-8.

5.5.6 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.

```
-- interface ORB {
package CORBA.ORB is

    -- ORB initialization
    type ORBid is new CORBA.String;

    package IDL_Sequence_String is
        new CORBA.Sequences.Unbounded( ...
    type Arg_List is new IDL_Sequence_String.Sequence;
    function Command_Line_Arguments return Arg_List;
    procedure Init
                (ORB_Identifier : in ORBid;
                 Argv           : in Arg_List
                                  := Command_Line_Arguments);

    type ObjectID is new CORBA.String;

    package ObjectId_Unbounded is
        new CORBA.Sequences.Unbounded ( ...
    type ObjectIdList is new ObjectId_Unbounded.Sequence;

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

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

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

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

-- Dynamic Invocation related operations
procedure Create_List
         (Count    : in     CORBA.Long;
          New_List :    out CORBA.NVList.Ref);

procedure Create_Operation_List
         (Oper     : in     CORBA.OperationDef.Ref;
          New_List :    out CORBA.NVList.Ref);

function Get_Default_Context return CORBA.Context.Ref;
procedure Get_Default_Context
    (The_Default_Value : out CORBA.Context.Ref);

package Request_Unbounded is
    new CORBA.Sequences.Unbounded( ...
type RequestSeq is new Request_Unbounded.Sequence;

procedure Send_Multiple_Requests_Oneway
    (Req : in RequestSeq);
procedure Send_Multiple_Requests_Deferred
    (Req : in RequestSeq);
function Poll_Next_Response return Boolean;
procedure Get_Next_Response
    (Req : out CORBA.Request.Object);

-- Service information operations
type ServiceType is new Unsigned_Short;
type ServiceOption is new Unsigned_Long;
type ServiceDetailType is new Unsigned_Long;

Security : constant ServiceType := 1;

type ServiceDetail is
  record
      service_detail_type : ServiceDetailType;
      service_detail      : CORBA.String;
  end record;

package IDL_SEQUENCE_ServiceOption is
    new CORBA.Sequences.Unbounded( ...
package IDL_SEQUENCE_ServiceDetail is
   new CORBA.Sequences.Unbounded( ...
type ServiceInformation is record
  service_options : IDL_SEQUENCE_ServiceOption.Sequence;
  service_details : IDL_SEQUENCE_ServiceDetail.Sequence;
end record;
```

```
procedure get_service_information
   (service_type        : in  ServiceType;
    service_information : out ServiceInformation;
    Returns             : out CORBA.Boolean);

-- initial reference operation
function List_Initial_Services return ObjectIDList;

function Resolve_Initial_References
    (Identifier : in ObjectID)
    return CORBA.Object.Ref'CLASS;
function Resolve_Initial_References
    (Identifier : in Standard.String)
    return CORBA.Object.Ref'CLASS;

-- TypeCode creation in CORBA.ORB.TypeCode_Creation

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

-- policy related operations
type PolicyErrorCode is new CORBA.Short;
BAD_POLICY               : constant PolicyErrorCode
                         := PolicyErrorCode'(0);
UNSUPPORTED_POLICY       : constant PolicyErrorCode
                         := PolicyErrorCode'(1);
BAD_POLICY_TYPE          : constant PolicyErrorCode
                         := PolicyErrorCode'(2);
BAD_POLICY_VALUE         : constant PolicyErrorCode
                         := PolicyErrorCode'(3);
UNSUPPORTED_POLICY_VALUE : constant PolicyErrorCode
                         := PolicyErrorCode'(4);

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

function Create_Policy(Of_Type : PolicyType; Val : Any)
    return CORBA.Policy.Ref;

-- there is no need in Ada for value factories
```

```
end CORBA.ORB;
```

The **TypeCode** creation functions have been moved to a child package **CORBA.ORB.TypeCode**.

5.5.7 Current

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

```
package CORBA.Current is

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

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

5.5.8 Policy

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

```
package CORBA.Policy is

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

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


   package IDL_SEQUENCE_Policy is new
      CORBA.Sequences.Unbounded
```

```
      (Ref);
   type PolicyList is new IDL_SEQUENCE_Policy.Sequence;


private

   ... implementation defined ...

end CORBA.Policy;
```

5.5.9 DomainManager

The domain manager provides mechanisms for:

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

```
package CORBA.DomainManager is

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

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

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

end CORBA.DomainManager;
```

5.5.10 ConstructionPolicy

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

```
package CORBA.ConstructionPolicy is

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

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

end CORBA.ConstructionPolicy;
```

5.6 Ada Specific Support packages

The mapping for Ada requires a number of packages in order to complete the mapping.

5.6.1 CORBA.Forward

The **CORBA.Forward** package is instantiated for every forward interface declaration. Its definition has been presented in Section 4.9.1, “Forward Definition Packages,” on page 4-22.

5.6.2 CORBA.Value_Forward

The **CORBA.Value_Forward** package is instantiated for every forward interface declaration. Its definition has been presented in Section 4.9.1, “Forward Definition Packages,” on page 4-22.

5.6.3 CORBA.Value.Box

The **CORBA.Value.Box** package is instantiated for every value box declaration. Its definition has been presented in Section 4.10.1, “Value Box Package,” on page 4-25.

5.6.4 CORBA.Iterate_Over_Any_Elements

CORBA.Iterate_Over_Any_Elements package aids in the analysis and decomposition of a type **any** for a type for which no IDL is known to exist. Its definition has been presented in Section 3.15.2, “Handling Unknown Types,” on page 3-13.

5.6.5 CORBA.Bounded_Strings and CORBA.Bounded_Wide_Strings

As explained in Section 3.9, “Mapping for String Types,” on page 3-8 and Section 3.10, “Mapping for Wide String Types,” on page 3-9, conforming products must supply substitute packages for **Ada.Strings.Bounded.Generic_Bounded_Length** and **Ada.Strings.Wide_Bounded.Generic_Bounded_Length**.

5.6.6 CORBA.Sequences

This mapping defines three packages to aid in the mapping of sequences: **CORBA.Seqeunces**, **CORBA.Sequences.Unbounded**, and **CORBA.Sequences.Bounded**. Conforming implementations shall provide implementations with semantics that mimic those of the similar named subprograms in the **Ada.Strings** package definitions. The specification of these packages is:

```
package CORBA.Sequences is

    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;


generic
    ...
package CORBA.Sequences.Unbounded is

    subtype Index_Range is Index_Type
        range 1 .. Index_Type'Last;
    subtype Length_Range is Index_Type'BASE
        range 0 .. Index_Type'Last;


    Null_Element_Array : Element_Array(1..0);

    type Sequence is private;

    Null_Sequence : constant Sequence;
        -- initial value of all Sequences

    function Length (Source : in Sequence) return Natural;

    type Element_Array_Access is access all Element_Array;

    procedure Free is new Ada.Unchecked_Deallocation
       (Element_Array, Element_Array_Access);

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

    function To_Sequence(Source : in Element_Array)
        return Sequence;
    procedure Set(Item : in out Sequence;
                  Source : in Element_Array);

    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 Get_Element
    (Source : in Sequence;
     Index  : in Index_Range)
     return   Element;

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

function Slice
    (Source : in Sequence;
     Low    : in Index_Range;
     High   : in Index_Type)
     return   Element_Array;

function "=" (Left, Right : in Sequence)
```

```
    return Standard.Boolean;

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

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

function Is_Null(Source : in Sequence)
    return Standard.Boolean;
-- equivalent to (Source = Null_Sequence)

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

function Index
    (Source   : in Sequence;
     Pattern  : in Element_Array;
     Going    : in Direction := Forward)
     return     Length_Range;  -- 0 = not found

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

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

function Replace_Slice
    (Source : in Sequence;
     Low    : in Index_Range;
     High   : in Index_Type;
     By     : in Element_Array)
     return   Sequence;

procedure Replace_Slice
    (Source   : in out Sequence;
     Low      : in Index_Range;
     High     : in Index_Type;
     By       : in Element_Array);

function Insert
    (Source   : in Sequence;
     Before   : in Index_Range;
```

```
     New_Item : in Element_Array)
     return     Sequence;

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

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

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

function Delete
    (Source  : in Sequence;
     From    : in Index_Range;
     Through : in Index_Type)
     return    Sequence;

procedure Delete
    (Source  : in out Sequence;
     From    : in Index_Range;
     Through : in Index_Type);

-----------------------------------
-- 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;

    end CORBA.Sequences.Unbounded;


    generic
        ...
    package CORBA.Sequences.Bounded is

        subtype Index_Range is Index_Type
            range 1 .. Index_Type'Last;
        subtype Length_Range is Index_Type'BASE
            range 0 .. Index_Type'Last;
        Max_Length : constant Index_Range := Index_Range(Max);

        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 is new Ada.Unchecked_Deallocation
           ( Element_Array, 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 Index_Range)
       return Sequence;

   procedure Set
       (Item   : in out Sequence;
        Source : in     Element_Array;
        Drop   : in     Truncation    := Error);

   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 Get_Element
    (Source : in Sequence;
     Index  : in Index_Range)
     return   Element;

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

function Slice
    (Source : in Sequence;
     Low    : in Index_Range;
     High   : in Index_Type)
     return   Element_Array;


function "=" (Left, Right : in Sequence)
```

```
    return Standard.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     Length_Range; -- 0 indicates not found

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

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

function Replace_Slice
    (Source : in Sequence;
     Low    : in Index_Range;
     High   : in Index_Type;
     By     : in Element_Array;
     Drop   : in Truncation    := Error)
     return   Sequence;

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


function Insert
    (Source   : in Sequence;
     Before   : in Index_Range;
     New_Item : in Element_Array;
```

```
     Drop     : in Truncation    := Error)
     return     Sequence;

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

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

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

function Delete
    (Source  : in Sequence;
     From    : in Index_Range;
     Through : in Index_Type)
     return    Sequence;

procedure Delete
    (Source  : in out Sequence;
     From    : in     Index_Range;
     Through : in     Index_Type);

-----------------------------------
-- 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;

    end CORBA.Sequences.Bounded;
```

References A

A.1 List of References

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

1. IDL Type Extensions RFP, March 1995. OMG TC Document 95-1-35.
2. The Common Object Request Broker: Architecture and Specification, Revision 2.2, February 1998.
3. CORBAservices: Common Object Services Specification, Revised Edition, OMG TC Document 95-3-31.
4. COBOL Language Mapping RFP, December 1995. OMG TC document 95-12-10.
5. COBOL 85 ANSI X3.23-1985 / ISO 1989-1985.
6. IEEE Standard for Binary Floating-Point Arithmetic, ANIS/IEEE Std 754-1985.
7. XDR: External Data Representation Standard, RFC1832, R. Srinivasan, Sun Microsystems, August 1995.
8. OSF Character and Code Set Registry, OSF DCE SIG RFC 40.1 (Public Version), S. (Martin) O'Donnell, June 1994.
9. RPC Runtime Support For I18N Characters — Functional Specification, OSF DCE SIG RFC 41.2, M. Romagna, R. Mackey, November 1994.
10. X/Open System Interface Definitions, Issue 4 Version 2, 1995.
11. ISO/IEC 862:1995 "Information Technology -- Programming Languages -- Ada"

Glossary

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

Class

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

Class-wide types

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

Controlled type

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

Package

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

Primitive operations

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

Subsystems

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

Tagged type

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

Withing, withs, with clause

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

Index

Index