The Interface Repository 8

Contents

This chapter contains the following sections.

8.1 Overview

The Interface Repository is the component of the ORB that provides persistent storage of interface definitions—it manages and provides access to a collection of object definitions specified in OMG IDL.

An ORB provides distributed access to a collection of objects using the objects' publicly defined interfaces specified in OMG IDL. The Interface Repository provides for the storage, distribution, and management of a collection of related objects' interface definitions.

For an ORB to correctly process requests, it must have access to the definitions of the objects it is handling. Object definitions can be made available to an ORB in one of two forms:

1. By incorporating the information procedurally into stub routines (e.g., as code that maps C language subroutines into communication protocols).
2. As objects accessed through the dynamically accessible Interface Repository (i.e., as interface objects" accessed through OMG IDL-specified interfaces).

In particular, the ORB can use object definitions maintained in the Interface Repository to interpret and handle the values provided in a request to:

- Provide type-checking of request signatures (whether the request was issued through the DII or through a stub).
- Assist in checking the correctness of interface inheritance graphs.
- Assist in providing interoperability between different ORB implementations.

As the interface to the object definitions maintained in an Interface Repository is public, the information maintained in the Repository can also be used by clients and services. For example, the Repository can be used to:

- Manage the installation and distribution of interface definitions.
- Provide components of a CASE environment (for example, an interface browser).
- Provide interface information to language bindings (such as a compiler).
- Provide components of end-user environments (for example, a menu bar constructor).

The complete OMG IDL specification for the Interface Repository is in Section 8.8, "OMG IDL for Interface Repository," on page 8-44; however, fragments of the specification are used throughout this chapter as necessary.

8.2 Scope of an Interface Repository

Interface definitions are maintained in the Interface Repository as a set of objects that are accessible through a set of OMG IDL-specified interface definitions. An interface definition contains a description of the operations it supports, including the types of the parameters, exceptions it may raise, and context information it may use.

In addition, the interface repository stores constant values, which might be used in other interface definitions or might simply be defined for programmer convenience and it stores typecodes, which are values that describe a type in structural terms.

The Interface Repository uses modules as a way to group interfaces and to navigate through those groups by name. Modules can contain constants, typedefs, exceptions, interface definitions, and other modules. Modules may, for example, correspond to the organization of OMG IDL definitions. They may also be used to represent organizations defined for administration or other purposes.

The Interface Repository is a set of objects that represent the information in it. There are operations that operate on this apparent object structure. It is an implementation's choice whether these objects exist persistently or are created when referenced in an operation on the repository. There are also operations that extract information in an efficient form, obtaining a block of information that describes a whole interface or a whole operation.

An ORB may have access to multiple Interface Repositories. This may occur because

- two ORBs have different requirements for the implementation of the Interface Repository,
- an object implementation (such as an OODB) prefers to provide its own type information, or
- it is desired to have different additional information stored in different repositories.

The use of typecodes and repository identifiers is intended to allow different repositories to keep their information consistent.

As shown in Figure 8-1 on page 8-4, the same interface **Doc** is installed in two different repositories, one at SoftCo, Inc., which sells Doc objects, and one at Customer, Inc., which buys Doc objects from SoftCo. SoftCo sets the repository id for the Doc interface when it defines it. Customer might first install the interface in its repository in a module where it could be tested before exposing it for general use. Because it has the same repository id, even though the Doc interface is stored in a different repository and is nested in a different module, it is known to be the same.

Meanwhile at SoftCo, someone working on a new Doc interface has given it a new repository id 456, which allows the ORBs to distinguish it from the current product Doc interface.

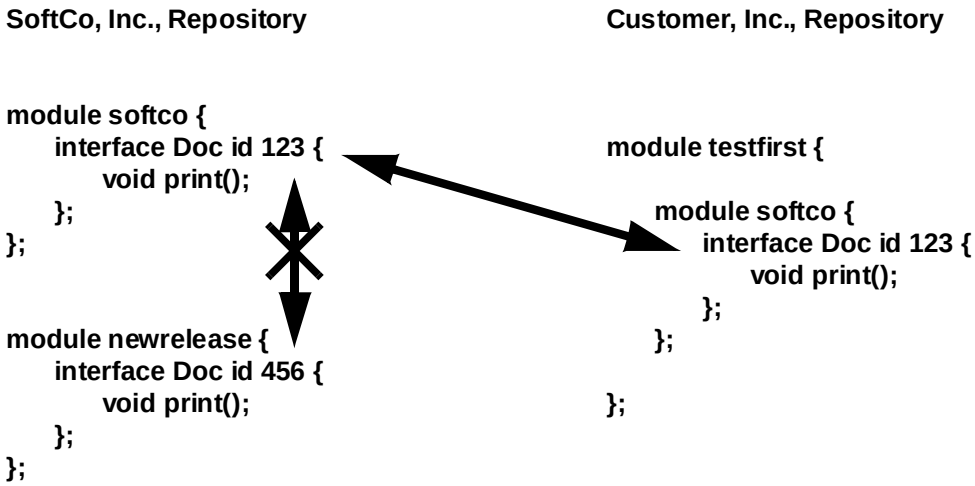

Figure 8-1 Using Repository IDs to establish correspondence between repositories

Not all interfaces will be visible in all repositories. For example, Customer employees cannot see the new release of the Doc interface. However, widely used interfaces will generally be visible in most repositories.

This Interface Repository specification defines operations for retrieving information from the repository as well as creating definitions within it. There may be additional ways to insert information into the repository (for example, compiling OMG IDL definitions, copying objects from one repository to another, etc.).

A critical use of the interface repository information is for connecting ORBs together. When an object is passed in a request from one ORB to another, it may be necessary to create a new object to represent the passed object in the receiving ORB. This may require locating the interface information in an interface repository in the receiving ORB. By getting the repository id from a repository in the sending ORB, it is possible to look up the interface in a repository in the receiving ORB. To succeed, the interface for that object must be installed in both repositories with the same repository id.

8.3 Implementation Dependencies

An implementation of an Interface Repository requires some form of persistent object store. Normally the kind of persistent object store used determines how interface definitions are distributed and/or replicated throughout a network domain. For example, if an Interface Repository is implemented using a filing system to provide object storage, there may be only a single copy of a set of interfaces maintained on a single machine. Alternatively, if an OODB is used to provide object storage, multiple copies of interface definitions may be maintained each of which is distributed across several machines to provide both high-availability and load-balancing.

The kind of object store used may determine the scope of interface definitions provided by an implementation of the Interface Repository. For example, it may determine whether each user has a local copy of a set of interfaces or if there is one copy per community of users. The object store may also determine whether or not all clients of an interface set see exactly the same set at any given point in time or whether latency in distributing copies of the set gives different users different views of the set at any point in time.

An implementation of the Interface Repository is also dependent on the security mechanism in use. The security mechanism (usually operating in conjunction with the object store) determines the nature and granularity of access controls available to constrain access to objects in the repository.

8.3.1 Managing Interface Repositories

Interface Repositories contain the information necessary to allow programs to determine and manipulate the type information at run-time. Programs may attempt to access the interface repository at any time by using the **get_interface** operation on the object reference. Once information has been installed in the repository, programs, stubs, and objects may depend on it. Updates to the repository must be done with care to avoid disrupting the environment. A variety of techniques are available to help do so.

A coherent repository is one whose contents can be expressed as a valid collection of OMG IDL definitions. For example, all inherited interfaces exist, there are no duplicate operation names or other name collisions, all parameters have known types, and so forth. As information is added to the repository, it is possible that it may pass through incoherent states. Media failures or communication errors might also cause it to appear incoherent. In general, such problems cannot be completely eliminated.

Replication is one technique to increase the availability and performance of a shared database. It is likely that the same interface information will be stored in multiple repositories in a computing environment. Using repository IDs, the repositories can establish the identity of the interfaces and other information across the repositories.

Multiple repositories might also be used to insulate production environments from development activity. Developers might be permitted to make arbitrary updates to their repositories, but administrators may control updates to widely used repositories. Some repository implementations might permit sharing of information, for example, several developers' repositories may refer to parts of a shared repository. Other repository implementations might instead copy the common information. In any case, the result should be a repository facility that creates the impression of a single, coherent repository.

The interface repository itself cannot make all repositories have coherent information, and it may be possible to enter information that does not make sense. The repository will report errors that it detects (e.g., defining two attributes with the same name) but might not report all errors, for example, adding an attribute to a base interface may or may not detect a name conflict with a derived interface. Despite these limitations, the

expectation is that a combination of conventions, administrative controls, and tools that add information to the repository will work to create a coherent view of the repository information.

Transactions and concurrency control mechanisms defined by the Object Services may be used by some repositories when updating the repository. Those services are designed so that they can be used without changing the operations that update the repository. For example, a repository that supports the Transaction Service would inherit the Repository interface, which contains the update operations, as well as the Transaction interface, which contains the transaction management operations. (For more information about Object Services, including the Transaction and Concurrency Control Services, refer to *CORBAservices: Common Object Service Specifications*.)

Often, rather than change the information, new versions will be created, allowing the old version to continue to be valid. The new versions will have distinct repository IDs and be completely different types as far as the repository and the ORBs are concerned. The IR provides storage for version identifiers for named types, but does not specify any additional versioning mechanism or semantics.

8.4 Basics

This section introduces some basic ideas that are important to understanding the Interface Repository. Topics addressed in this section are:

- Names and IDs
- Types and TypeCodes
- Interface Objects

8.4.1 Names and Identifiers

Simple names are not necessarily unique within an Interface Repository; they are always relative to an explicit or implicit module. In this context, interface definitions are considered explicit modules.

Scoped names uniquely identify modules, interfaces, constant, typedefs, exceptions, attributes, and operations in an Interface Repository.

Repository identifiers globally identify modules, interfaces, constants, typedefs, exceptions, attributes, and operations. They can be used to synchronize definitions across multiple ORBs and Repositories.

8.4.2 Types and TypeCodes

The Interface Repository stores information about types that are not interfaces in a data value called a TypeCode. From the TypeCode alone it is possible to determine the complete structure of a type. See “TypeCodes” on page 8-35 for more information on the internal structure of TypeCodes.

8.4.3 Interface Objects

Each interface managed in an Interface Repository is maintained as a collection of interface objects:

- Repository: the top-level module for the repository name space; it contains constants, typedefs, exceptions, interface definitions, and modules.
- ModuleDef: a logical grouping of interfaces; it contains constants, typedefs, exceptions, interface definitions, and other modules.
- InterfaceDef: an interface definition; it contains lists of constants, types, exceptions, operations, and attributes.
- AttributeDef: the definition of an attribute of the interface.
- OperationDef: the definition of an operation on the interface; it contains lists of parameters and exceptions raised by this operation.
- TypedefDef: base interface for definitions of named types that are not interfaces.
- ConstantDef: the definition of a named constant.
- ExceptionDef: the definition of an exception that can be raised by an operation.

The interface specifications for each interface object lists the attributes maintained by that object (see “Interface Repository Interfaces” on page 8-9). Many of these attributes correspond directly to OMG IDL statements. An implementation can choose to maintain additional attributes to facilitate managing the Repository or to record additional (proprietary) information about an interface. Implementations that extend the IR interfaces should do so by deriving new interfaces, not by modifying the standard interfaces.

The *CORBA* specification defines a minimal set of operations for interface objects. Additional operations that an implementation of the Interface Repository may provide could include operations that provide for the versioning of interfaces and for the reverse compilation of specifications (i.e., the generation of a file containing an object’s OMG IDL specification).

8.4.4 Structure and Navigation of Interface Objects

The definitions in the Interface Repository are structured as a set of objects. The objects are structured the same way definitions are structured—some objects (definitions) “contain” other objects.

The containment relationships for the objects in the Interface Repository are shown in Figure 8-2 on page 8-8.

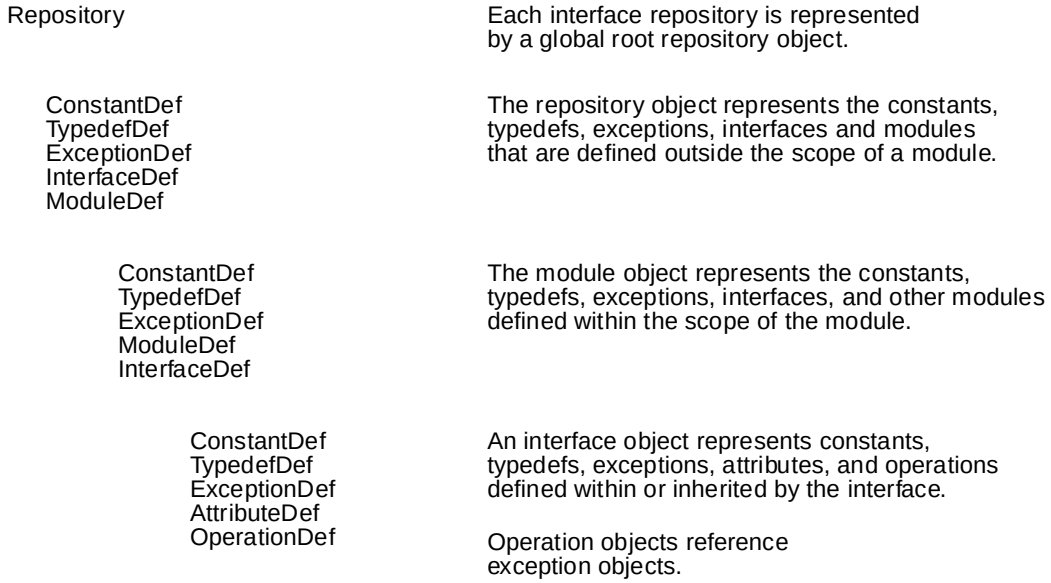

Figure 8-2 Interface Repository Object Containment

There are three ways to locate an interface in the Interface Repository, by:

1. Obtaining an **InterfaceDef** object directly from the ORB.
2. Navigating through the module name space using a sequence of names.
3. Locating the **InterfaceDef** object that corresponds to a particular repository identifier.

Obtaining an **InterfaceDef** object directly is useful when an object is encountered whose type was not known at compile time. By using the **get_interface()** operation on the object reference, it is possible to retrieve the Interface Repository information about the object. That information could then be used to perform operations on the object.

Navigating the module name space is useful when information about a particular named interface is desired. Starting at the root module of the repository, it is possible to obtain entries by name.

Locating the **InterfaceDef** object by ID is useful when looking for an entry in one repository that corresponds to another. A repository identifier must be globally unique. By using the same identifier in two repositories, it is possible to obtain the interface identifier for an interface in one repository, and then obtain information about that interface from another repository that may be closer or contain additional information about the interface.

8.5 Interface Repository Interfaces

Several abstract interfaces are used as base interfaces for other objects in the IR.

A common set of operations is used to locate objects within the Interface Repository. These operations are defined in the abstract interfaces **IRObject**, **Container**, and **Contained** described below. All IR objects inherit from the **IRObject** interface, which provides an operation for identifying the actual type of the object. Objects that are containers inherit navigation operations from the **Container** interface. Objects that are contained by other objects inherit navigation operations from the **Contained** interface.

The **IDLType** interface is inherited by all IR objects that represent IDL types, including interfaces, typedefs, and anonymous types. The **TypedefDef** interface is inherited by all named non-interface types.

The **IRObject**, **Contained**, **Container**, **IDLType**, and **TypedefDef** interfaces are not instantiable.

All string data in the Interface Repository are encoded as defined by the ISO 8859-1 coded character set.

8.5.1 Supporting Type Definitions

Several types are used throughout the IR interface definitions.

```
module CORBA {
typedef string          Identifier;
typedef string          ScopedName;
typedef string          RepositoryId;

enum DefinitionKind {
    dk_none, dk_all,
    dk_Attribute, dk_Constant, dk_Exception, dk_Interface,
    dk_Module, dk_Operation, dk_Typedef,
    dk_Alias, dk_Struct, dk_Union, dk_Enum,
    dk_Primitive, dk_String, dk_Sequence, dk_Array,
    dk_Repository,
    dk_Wstring, dk_Fixed
};
};
```

Identifiers are the simple names that identify modules, interfaces, constants, typedefs, exceptions, attributes, and operations. They correspond exactly to OMG IDL identifiers. An **Identifier** is not necessarily unique within an entire Interface Repository; it is unique only within a particular **Repository, ModuleDef**, **InterfaceDef**, or **OperationDef**.

A **ScopedName** is a name made up of one or more **Identifier**s separated by the characters “::”. They correspond to OMG IDL scoped names.

An *absolute* **ScopedName** is one that begins with "::" and unambiguously identifies a definition in a **Repository**. An *absolute* **ScopedName** in a **Repository** corresponds to a *global name* in an OMG IDL file. A *relative* **ScopedName** does not begin with "::" and must be resolved relative to some context.

A **RepositoryId** is an identifier used to uniquely and globally identify a module, interface, constant, typedef, exception, attribute or operation. As **RepositoryId**s are defined as strings, they can be manipulated (e.g., copied and compared) using a language binding's string manipulation routines.

A **DefinitionKind** identifies the type of an IR object.

8.5.2 IRObject

The **IRObject** interface represents the most generic interface from which all other Interface Repository interfaces are derived, even the Repository itself.

```
module CORBA {
    interface IRObject {
        // read interface
        readonly    attribute DefinitionKind     def_kind;

        // write interface
        void destroy ();
    };
};
```

Read Interface

The **def_kind** attribute identifies the type of the definition.

Write Interface

The **destroy** operation causes the object to cease to exist. If the object is a **Container**, **destroy** is applied to all its contents. If the object contains an **IDLType** attribute for an anonymous type, that **IDLType** is destroyed. If the object is currently contained in some other object, it is removed. Invoking **destroy** on a **Repository** or on a **PrimitiveDef** is an error. Implementations may vary in their handling of references to an object that is being destroyed, but the Repository should not be left in an incoherent state.

8.5.3 Contained

The **Contained** interface is inherited by all Interface Repository interfaces that are contained by other IR objects. All objects within the Interface Repository, except the root object (**Repository**) and definitions of anonymous (**ArrayDef**, **StringDef**, and **SequenceDef**), and primitive types are contained by other objects.

```
module CORBA {
typedef string VersionSpec;

interface Contained : IRObject {
    // read/write interface

            attribute RepositoryId          id;
            attribute Identifier            name;
            attribute VersionSpec           version;

    // read interface

    readonly    attribute Container             defined_in;
    readonly    attribute ScopedName            absolute_name;
    readonly    attribute Repository            containing_repository;

    struct Description {
        DefinitionKind      kind;
        any                 value;
    };

    Description describe ();

    // write interface
    void move (
        in Container        new_container,
        in Identifier       new_name,
        in VersionSpec      new_version
        );
    };
};
```

Read Interface

An object that is contained by another object has an **id** attribute that identifies it globally, and a **name** attribute that identifies it uniquely within the enclosing **Container** object. It also has a **version** attribute that distinguishes it from other versioned objects with the same **name**. IRs are not required to support simultaneous containment of multiple versions of the same named object. Supporting multiple versions most likely requires mechanism and policy not specified in this document.

Contained objects also have a **defined_in** attribute that identifies the **Container** within which they are defined. Objects can be contained either because they are defined within the containing object (for example, an interface is defined within a module) or because they are inherited by the containing object (for example, an operation may be contained by an interface because the interface inherits the operation from another interface). If an object is contained through inheritance, the **defined_in** attribute identifies the **InterfaceDef** from which the object is inherited.

The **absolute_name** attribute is an absolute **ScopedName** that identifies a **Contained** object uniquely within its enclosing **Repository**. If this object's **defined_in** attribute references a **Repository**, the **absolute_name** is formed by concatenating the string "::" and this object's **name** attribute. Otherwise, the **absolute_name** is formed by concatenating the **absolute_name** attribute of the object referenced by this object's **defined_in** attribute, the string "::", and this object's **name** attribute.

The **containing_repository** attribute identifies the **Repository** that is eventually reached by recursively following the object's **defined_in** attribute.

The **describe** operation returns a structure containing information about the interface. The description structure associated with each interface is provided below with the interface's definition. The kind of definition described by the structure returned is provided with the returned structure. For example, if the **describe** operation is invoked on an attribute object, the **kind** field contains **dk_Attribute** and the **value** field contains an **any**, which contains the **AttributeDescription** structure.

Write Interface

Setting the **id** attribute changes the global identity of this definition. An error is returned if an object with the specified **id** attribute already exists within this object's **Repository**.

Setting the **name** attribute changes the identity of this definition within its **Container**. An error is returned if an object with the specified **name** attribute already exists within this object's **Container**. The **absolute_name** attribute is also updated, along with any other attributes that reflect the name of the object. If this object is a **Container**, the **absolute_name** attribute of any objects it contains are also updated.

The **move** operation atomically removes this object from its current **Container**, and adds it to the **Container** specified by **new_container**, which must:

- Be in the same **Repository,**
- Be capable of containing this object's type (see "Structure and Navigation of Interface Objects" on page 8-7); and
- Not already contain an object with this object's name (unless multiple versions are supported by the IR).

The **name** attribute is changed to **new_name**, and the **version** attribute is changed to **new_version**.

The **defined_in** and **absolute_name** attributes are updated to reflect the new container and **name**. If this object is also a **Container**, the **absolute_name** attributes of any objects it contains are also updated.

8.5.4 Container

The **Container** interface is used to form a containment hierarchy in the Interface Repository. A **Container** can contain any number of objects derived from the **Contained** interface. All **Container**s, except for **Repository**, are also derived from **Contained**.

```
module CORBA {
typedef sequence <Contained> ContainedSeq;

interface Container : IRObject {
    // read interface

Contained lookup (in ScopedName search_name);

ContainedSeq contents (
    in DefinitionKind       limit_type,
    in boolean              exclude_inherited
);

ContainedSeq lookup_name (
    in Identifier           search_name,
    in long                 levels_to_search,
    in DefinitionKind       limit_type,
    in boolean              exclude_inherited
);
struct Description {
    Contained           contained_object;
    DefinitionKind      kind;
    any                 value;
};

typedef sequence<Description> DescriptionSeq;

DescriptionSeq describe_contents (
    in DefinitionKind       limit_type,
    in boolean              exclude_inherited,
    in long                 max_returned_objs
);

// write interface
```

```
ModuleDef create_module (
    in RepositoryId         id,
    in Identifier           name,
    in VersionSpec          version
);

ConstantDef create_constant (
    in RepositoryId         id,
    in Identifier           name,
    in VersionSpec          version,
    in IDLType              type,
    in any                  value
);

StructDef create_struct (
in RepositoryId         id,
in Identifier           name,
in VersionSpec          version,
in StructMemberSeq      members
);

UnionDef create_union (
in RepositoryId         id,
in Identifier           name,
in VersionSpec          version,
in IDLType              discriminator_type,
in UnionMemberSeq       members
);

EnumDef create_enum (
in RepositoryId         id,
in Identifier           name,
in VersionSpec          version,
in EnumMemberSeq        members
);

AliasDef create_alias (
in RepositoryId         id,
in Identifier           name,
in VersionSpec          version,
in IDLType              original_type
);

InterfaceDef create_interface (
in RepositoryId         id,
in Identifier           name,
in VersionSpec          version,
in InterfaceDefSeq      base_interfaces
);
```

```
            ExceptionDef create_exception(
            in RepositoryId id,
            in Identifier name,
            in VersionSpec version,
            in StructMemberSeq members
            );
    };
};
```

Read Interface

The **lookup** operation locates a definition relative to this container given a scoped name using OMG IDL's name scoping rules. An absolute scoped name (beginning with "::") locates the definition relative to the enclosing **Repository**. If no object is found, a nil object reference is returned.

The **contents** operation returns the list of objects directly contained by or inherited into the object. The operation is used to navigate through the hierarchy of objects. Starting with the Repository object, a client uses this operation to list all of the objects contained by the Repository, all of the objects contained by the modules within the Repository, and then all of the interfaces within a specific module, and so on.

limit_type	If **limit_type** is set to **dk_all**, objects of all interface types are returned. For example, if this is an **InterfaceDef**, the attribute, operation, and exception objects are all returned. If **limit_type** is set to a specific interface, only objects of that interface type are returned. For example, only attribute objects are returned if **limit_type** is set to **dk_Attribute**.
exclude_inherited	If set to TRUE, inherited objects (if there are any) are not returned. If set to FALSE, all contained objects—whether contained due to inheritance or because they were defined within the object—are returned. The **lookup_name** operation is used to locate an object by name within a particular object or within the objects contained by that object.
search_name	Specified which name is to be searched for.
levels_to_search	Controls whether the lookup is constrained to the object the operation is invoked on or whether it should search through objects contained by the object as well. Setting **levels_to_search** to -1 searches the current object and all contained objects. Setting **levels_to_search** to 1 searches only the current object.

limit_type	If **limit_type** is set to **dk_all**, objects of all interface types are returned (e.g., attributes, operations, and exceptions are all returned). If **limit_type** is set to a specific interface, only objects of that interface type are returned. For example, only attribute objects are returned if **limit_type** is set to **dk_Attribute**.
exclude_inherited	If set to TRUE, inherited objects (if there are any) are not returned. If set to FALSE, all contained objects (whether contained due to inheritance or because they were defined within the object) are returned. The **describe_contents** operation combines the **contents** operation and the **describe** operation. For each object returned by the **contents** operation, the description of the object is returned (i.e., the object's **describe** operation is invoked and the results returned).
max_returned_objs	Limits the number of objects that can be returned in an invocation of the call to the number provided. Setting the parameter to -1 means return all contained objects.

Write Interface

The **Container** interface provides operations to create **ModuleDef**s, **ConstantDef**s, **StructDef**s, **UnionDef**s, **EnumDef**s, **AliasDef**s, and **InterfaceDef**s as contained objects. The **defined_in** attribute of a definition created with any of these operations is initialized to identify the **Container** on which the operation is invoked, and the **containing_repository** attribute is initialized to its **Repository**.

The **create_<type>** operations all take **id** and **name** parameters which are used to initialize the identity of the created definition. An error is returned if an object with the specified **id** already exists within this object's **Repository**, or, assuming multiple versions are not supported, if an object with the specified **name** already exists within this **Container**.

The **create_module** operation returns a new empty **ModuleDef**. Definitions can be added using **Container::create_<type>** operations on the new module, or by using the **Contained::move** operation.

The **create_constant** operation returns a new **ConstantDef** with the specified **type** and **value**.

The **create_struct** operation returns a new **StructDef** with the specified **members**. The **type** member of the **StructMember** structures is ignored, and should be set to **TC_void**. See "StructDef" on page 8-20 for more information.

The **create_union** operation returns a new **UnionDef** with the specified **discriminator_type** and **members**. The **type** member of the **UnionMember** structures is ignored, and should be set to **TC_void**. See “UnionDef” on page 8-21 for more information.

The **create_enum** operation returns a new **EnumDef** with the specified **members**. See “EnumDef” on page 8-22 for more information.

The **create_alias** operation returns a new **AliasDef** with the specified **original_type**.

The **create_interface** operation returns a new empty **InterfaceDef** with the specified **base_interfaces**. Type, exception, and constant definitions can be added using **Container::create_<type>** operations on the new **InterfaceDef**. **OperationDefs** can be added using **InterfaceDef::create_operation** and **AttributeDefs** can be added using **Interface::create_attribute**. Definitions can also be added using the **Contained::move** operation.

The **create_exception** operation returns a new **ExceptionDef** with the specified members. The **type** member of the **StructMember** structures is ignored, and should be set to **TC_void**.

8.5.5 IDLType

The **IDLType** interface is an abstract interface inherited by all IR objects that represent OMG IDL types. It provides access to the **TypeCode** describing the type, and is used in defining other interfaces wherever definitions of IDL types must be referenced.

```
module CORBA {
interface IDLType : IRObject {
readonly    attribute TypeCode      type;
};
};
```

The **type** attribute describes the type defined by an object derived from **IDLType**.

8.5.6 Repository

Repository is an interface that provides global access to the Interface Repository. The **Repository** object can contain constants, typedefs, exceptions, interfaces, and modules. As it inherits from **Container**, it can be used to look up any definition (whether globally defined or defined within a module or interface) either by **name** or by **id**.

There may be more than one Interface Repository in a particular ORB environment (although some ORBs might require that definitions they use be registered with a particular repository). Each ORB environment will provide a means for obtaining object references to the Repositories available within the environment.

```
module CORBA {
    interface Repository : Container {
        // read interface

        Contained lookup_id (in RepositoryId search_id);

        PrimitiveDef get_primitive (in PrimitiveKind kind);

        // write interface

        StringDef create_string (in unsigned long bound);

        WstringDef create_wstring(in unsigned long bound);

        SequenceDef create_sequence (
                in unsigned long    bound,
                in IDLType          element_type
        );

        ArrayDef create_array (
                in unsigned long    length,
                in IDLType          element_type
        );

        FixedDef create_fixed(
            in unsigned short digits,
            in short scale
        );
    };
};
```

Read Interface

The **lookup_id** operation is used to lookup an object in a **Repository** given its **RepositoryId**. If the **Repository** does not contain a definition for **search_id**, a nil object reference is returned.

The **get_primitive** operation returns a reference to a **PrimitiveDef** with the specified **kind** attribute. All **PrimitiveDef**s are immutable and owned by the **Repository**.

Write Interface

The three **create_<type>** operations create new objects defining anonymous types. As these interfaces are not derived from **Contained**, it is the caller's responsibility to invoke **destroy** on the returned object if it is not successfully used in creating a definition that is derived from **Contained**. Each anonymous type definition must be used in defining exactly one other object.

The **create_string** operation returns a new **StringDef** with the specified **bound**, which must be non-zero. The **get_primitive** operation is used for unbounded strings.

The **create_wstring** operation returns a new **WstringDef** with the specified **bound**, which must be non-zero. The **get_primitive** operation is used for unbounded strings.

The **create_sequence** operation returns a new **SequenceDef** with the specified **bound** and **element_type**.

The **create_array** operation returns a new **ArrayDef** with the specified **length** and **element_type**.

The **create_fixed** operation returns a new **FixedDef** with the specified number of digits and scale. The number of digits must be from 1 to 31, inclusive.

8.5.7 ModuleDef

A **ModuleDef** can contain constants, typedefs, exceptions, interfaces, and other module objects.

```
module CORBA {
    interface ModuleDef : Container, Contained {
    };

    struct ModuleDescription {
        Identifier      name;
        RepositoryId    id;
        RepositoryId    defined_in;
        VersionSpec     version;
    };
};
```

The inherited **describe** operation for a **ModuleDef** object returns a **ModuleDescription**.

8.5.8 ConstantDef Interface

A **ConstantDef** object defines a named constant.

```
module CORBA {
    interface ConstantDef : Contained {
        readonly    attribute TypeCode      type;
                    attribute IDLType       type_def;
                    attribute any           value;
    };

    struct ConstantDescription {
        Identifier          name;
        RepositoryId        id;
        RepositoryId        defined_in;
        VersionSpec         version;
        TypeCode            type;
        any                 value;
```

```
    };
};
```

Read Interface

The **type** attribute specifies the **TypeCode** describing the type of the constant. The type of a constant must be one of the simple types (long, short, float, char, string, octet, etc.). The **type_def** attribute identifies the definition of the type of the constant.

The **value** attribute contains the value of the constant, not the computation of the value (e.g., the fact that it was defined as “1+2”).

The **describe** operation for a **ConstantDef** object returns a **ConstantDescription**.

Write Interface

Setting the **type_def** attribute also updates the **type** attribute.

When setting the **value** attribute, the **TypeCode** of the supplied any must be equal to TypedefDef Interface

TypedefDef is an abstract interface used as a base interface for all named non-object types (structures, unions, enumerations, and aliases). The **TypedefDef** interface is not inherited by the definition objects for primitive or anonymous types.

```
module CORBA {
    interface TypedefDef : Contained, IDLType {
    };

    struct TypeDescription {
        Identifier      name;
        RepositoryId    id;
        RepositoryId    defined_in;
        VersionSpec     version;
        TypeCode        type;
    };
};
```

The inherited **describe** operation for interfaces derived from **TypedefDef** returns a **TypeDescription**.

8.5.9 StructDef

A **StructDef** represents an OMG IDL structure definition. It can contain structs, unions, and enums.

```
module CORBA {
    struct StructMember {
        Identifier      name;
        TypeCode        type;
```

```
        IDLType         type_def;
    };
    typedef sequence <StructMember> StructMemberSeq;

    interface StructDef : TypedefDef, Container{
        attribute StructMemberSeq       members;
    };
};
```

Read Interface

The **members** attribute contains a description of each structure member. It can contain structs, unions, and enums.

The inherited **type** attribute is a **tk_struct TypeCode** describing the structure.

Write Interface

Setting the **members** attribute also updates the **type** attribute. When setting the **members** attribute, the **type** member of the **StructMember** structure is ignored and should be set to **TC_void**.

8.5.10 UnionDef

A **UnionDef** represents an OMG IDL union definition. It can contain structs, unions, and enums.

```
module CORBA {
    struct UnionMember {
        Identifier      name;
        any             label;
        TypeCode        type;
        IDLType         type_def;
    };
    typedef sequence <UnionMember> UnionMemberSeq;

    interface UnionDef : TypedefDef, Container {
        readonly    attribute TypeCode          discriminator_type;
                    attribute IDLType           discriminator_type_def;
                    attribute UnionMemberSeq    members;
    };
};
```

Read Interface

The **discriminator_type** and **discriminator_type_def** attributes describe and identify the union's discriminator type.

The **members** attribute contains a description of each union member. The **label** of each **UnionMemberDescription** is a distinct value of the **discriminator_type**. Adjacent members can have the same **name**. Members with the same **name** must also have the same **type**. A **label** with type **octet** and value 0 indicates the default union member.

The inherited **type** attribute is a **tk_union TypeCode** describing the union.

Write Interface

Setting the **discriminator_type_def** attribute also updates the **discriminator_type** attribute and setting the **discriminator_type_def** or **members** attribute also updates the **type** attribute.

When setting the **members** attribute, the **type** member of the **UnionMember** structure is ignored and should be set to **TC_void**.

8.5.11 EnumDef

An **EnumDef** represents an OMG IDL enumeration definition.

```
module CORBA {
    typedef sequence <Identifier> EnumMemberSeq;

    interface EnumDef : TypedefDef {
        attribute EnumMemberSeq     members;
    };
};
```

Read Interface

The **members** attribute contains a distinct name for each possible value of the enumeration.

The inherited **type** attribute is a **tk_enum TypeCode** describing the enumeration.

Write Interface

Setting the **members** attribute also updates the **type** attribute.

8.5.12 AliasDef

An **AliasDef** represents an OMG IDL typedef that aliases another definition.

```
module CORBA {
    interface AliasDef : TypedefDef {
        attribute IDLType   original_type_def;
    };
};
```

Read Interface

The **original_type_def** attribute identifies the type being aliased.

The inherited **type** attribute is a **tk_alias TypeCode** describing the alias.

Write Interface

Setting the **original_type_def** attribute also updates the **type** attribute.

8.5.13 PrimitiveDef

A **PrimitiveDef** represents one of the OMG IDL primitive types. As primitive types are unnamed, this interface is not derived from **TypedefDef** or **Contained**.

```
module CORBA {
    enum PrimitiveKind {
        pk_null, pk_void, pk_short, pk_long, pk_ushort, pk_ulong,
        pk_float, pk_double, pk_boolean, pk_char, pk_octet,
        pk_any, pk_TypeCode, pk_Principal, pk_string, pk_objref,
        pk_longlong, pk_ulonglong, pk_longdouble, pk_wchar, pk_wstring
    };

    interface PrimitiveDef: IDLType {
    readonly attribute PrimitiveKind    kind;
    };
};
```

The **kind** attribute indicates which primitive type the **PrimitiveDef** represents. There are no **PrimitiveDef**s with kind **pk_null**. A **PrimitiveDef** with kind **pk_string** represents an unbounded string. A **PrimitiveDef** with kind **pk_objref** represents the IDL type **Object**.

The inherited **type** attribute describes the primitive type.

All **PrimitiveDef**s are owned by the Repository. References to them are obtained using **Repository::get_primitive**.

8.5.14 StringDef

A **StringDef** represents an IDL bounded string type. The unbounded string type is represented as a **PrimitiveDef**. As string types are anonymous, this interface is not derived from **TypedefDef** or **Contained**.

```
module CORBA {
    interface StringDef : IDLType {
        attribute unsigned long     bound;
    };
};
```

The **bound** attribute specifies the maximum number of characters in the string and must not be zero.

The inherited **type** attribute is a **tk_string TypeCode** describing the string.

8.5.15 WstringDef

A **WstringDef** represents an IDL wide string. The unbounded wide string type is represented as a **PrimitiveDef**. As wide string types are anonymous, this interface is not derived from **TypedefDef** or **Contained.**

```
module CORBA {
    interface WstringDef : IDLType {
        attribute unsigned long  bound;
    };
};
```

The **bound** attribute specifies the maximum number of wide characters in a wide string, and must not be zero.

The inherited **type** attribute is a **tk_wstring TypeCode** describing the wide string.

8.5.16 FixedDef

A **FixedDef** represents an IDL fixed point type.

```
module CORBA {
    interface FixedDef : IDLType {
        attribute unsigned short digits;
        attribute short scale;
    };
};
```

The **digits** attribute specifies the total number of decimal digits in the number, and must be from 1 to 31, inclusive. The **scale** attribute specifies the position of the decimal point.

The inherited **type** attribute is a **tk_fixed TypeCode**, which describes a fixed-point decimal number.

8.5.17 SequenceDef

A **SequenceDef** represents an IDL sequence type. As sequence types are anonymous, this interface is not derived from **TypedefDef** or **Contained**.

```
module CORBA {
    interface SequenceDef : IDLType {
                    attribute unsigned long     bound;
    readonly        attribute TypeCode          element_type;
                    attribute IDLType           element_type_def;
```

```
    };
};
```

Read Interface

The **bound** attribute specifies the maximum number of elements in the sequence. A **bound** of zero indicates an unbounded sequence.

The type of the elements is described by **element_type** and identified by **element_type_def**.

The inherited **type** attribute is a **tk_sequence TypeCode** describing the sequence.

Write Interface

Setting the **element_type_def** attribute also updates the **element_type** attribute.

Setting the **bound** or **element_type_def** attribute also updates the **type** attribute.

8.5.18 ArrayDef

An **ArrayDef** represents an IDL array type. As array types are anonymous, this interface is not derived from **TypedefDef** or **Contained**.

```
module CORBA {
    interface ArrayDef : IDLType {
                attribute unsigned long    length;
    readonly    attribute TypeCode         element_type;
                attribute IDLType          element_type_def;
    };
};
```

Read Interface

The **length** attribute specifies the number of elements in the array.

The type of the elements is described by **element_type** and identified by **element_type_def**. Since an **ArrayDef** only represents a single dimension of an array, multi-dimensional IDL arrays are represented by multiple **ArrayDef** objects, one per array dimension. The **element_type_def** attribute of the **ArrayDef** representing the leftmost index of the array, as defined in IDL, will refer to the **ArrayDef** representing the next index to the right, and so on. The innermost **ArrayDef** represents the rightmost index and the element type of the multi-dimensional OMG IDL array.

The inherited **type** attribute is a **tk_array TypeCode** describing the array.

Write Interface

Setting the **element_type_def** attribute also updates the **element_type** attribute.

Setting the **bound** or **element_type_def** attribute also updates the **type** attribute.

8.5.19 ExceptionDef

An **ExceptionDef** represents an exception definition. It can contain structs, unions, and enums.

```
module CORBA {
    interface ExceptionDef : Contained, Container {
        readonly    attribute TypeCode              type;
                    attribute StructMemberSeq       members;
    };

    struct ExceptionDescription {
        Identifier      name;
        RepositoryId    id;
        RepositoryId    defined_in;
        VersionSpec     version;
        TypeCode        type;
    };
};
```

Read Interface

The **type** attribute is a **tk_except TypeCode** describing the exception.

The members **attribute** describes any exception members.

The **describe** operation for a **ExceptionDef** object returns an **ExceptionDescription**.

Write Interface

Setting the **members** attribute also updates the **type** attribute. When setting the **members** attribute, the **type** member of the **StructMember** structure is ignored and should be set to **TC_void**.

8.5.20 AttributeDef

An **AttributeDef** represents the information that defines an attribute of an interface.

```
module CORBA {
    enum AttributeMode {ATTR_NORMAL, ATTR_READONLY};

    interface AttributeDef : Contained {
        readonly    attribute TypeCode          type;
                    attribute IDLType           type_def;
                    attribute AttributeMode     mode;
    };
```

```
    struct AttributeDescription {
        Identifier          name;
        RepositoryId        id;
        RepositoryId        defined_in;
        VersionSpec         version;
        TypeCode            type;
        AttributeMode       mode;
    };
};
```

Read Interface

The **type** attribute provides the **TypeCode** describing the type of this attribute. The **type_def** attribute identifies the object defining the type of this attribute.

The **mode** attribute specifies read only or read/write access for this attribute.

Write Interface

Setting the **type_def** attribute also updates the **type** attribute.

8.5.21 OperationDef

An **OperationDef** represents the information needed to define an operation of an interface.

```
module CORBA {
    enum OperationMode {OP_NORMAL, OP_ONEWAY};

    enum ParameterMode {PARAM_IN, PARAM_OUT, PARAM_INOUT};
    struct ParameterDescription {
        Identifier          name;
        TypeCode            type;
        IDLType             type_def;
        ParameterMode       mode;
    };
    typedef sequence <ParameterDescription> ParDescriptionSeq;

    typedef Identifier ContextIdentifier;
    typedef sequence <ContextIdentifier> ContextIdSeq;

    typedef sequence <ExceptionDef> ExceptionDefSeq;
    typedef sequence <ExceptionDescription> ExcDescriptionSeq;

    interface OperationDef : Contained {
        readonly    attribute TypeCode          result;
                    attribute IDLType           result_def;
                    attribute ParDescriptionSeq params;
                    attribute OperationMode     mode;
```

```
                attribute ContextIdSeq      contexts;
                attribute ExceptionDefSeq   exceptions;
    };

    struct OperationDescription {
        Identifier              name;
        RepositoryId            id;
        RepositoryId            defined_in;
        VersionSpec             version;
        TypeCode                result;
        OperationMode           mode;
        ContextIdSeq            contexts;
        ParDescriptionSeq       parameters;
        ExcDescriptionSeq       exceptions;
    };
};
```

Read Interface

The **result** attribute is a **TypeCode** describing the type of the value returned by the operation. The **result_def** attribute identifies the definition of the returned type.

The **params** attribute describes the parameters of the operation. It is a sequence of **ParameterDescription** structures. The order of the **ParameterDescription**s in the sequence is significant. The **name** member of each structure provides the parameter name. The **type** member is a **TypeCode** describing the type of the parameter. The **type_def** member identifies the definition of the type of the parameter. The **mode** member indicates whether the parameter is an in, out, or inout parameter.

The operation's **mode** is either oneway (i.e., no output is returned) or normal.

The **contexts** attribute specifies the list of context identifiers that apply to the operation.

The **exceptions** attribute specifies the list of exception types that can be raised by the operation.

The inherited **describe** operation for an **OperationDef** object returns an **OperationDescription**.

The inherited **describe_contents** operation provides a complete description of this operation, including a description of each parameter defined for this operation.

Write Interface

Setting the **result_def** attribute also updates the **result** attribute.

The **mode** attribute can only be set to **OP_ONEWAY** if the result is **TC_void** and all elements of **params** have a **mode** of **PARAM_IN**.

8.5.22 *InterfaceDef*

An **InterfaceDef** object represents an interface definition. It can contain constants, typedefs, exceptions, operations, and attributes.

```
module CORBA {
    interface InterfaceDef;
    typedef sequence <InterfaceDef> InterfaceDefSeq;
    typedef sequence <RepositoryId> RepositoryIdSeq;
    typedef sequence <OperationDescription> OpDescriptionSeq;
    typedef sequence <AttributeDescription> AttrDescriptionSeq;

    interface InterfaceDef : Container, Contained, IDLType {
        // read/write interface

        attribute InterfaceDefSeq       base_interfaces;

        // read interface

        boolean is_a (in RepositoryId interface_id);

        struct FullInterfaceDescription {
            Identifier                  name;
            RepositoryId                id;
            RepositoryId                defined_in;
            VersionSpec                 version;
            OpDescriptionSeq            operations;
            AttrDescriptionSeq          attributes;
            RepositoryIdSeq             base_interfaces;
            TypeCode                    type;
        };

        FullInterfaceDescription describe_interface();

        // write interface

        AttributeDef create_attribute (
                in RepositoryId         id,
                in Identifier           name,
                in VersionSpec          version,
                in IDLType              type,
                in AttributeMode        mode
        );

        OperationDef create_operation (
                in RepositoryId         id,
                in Identifier           name,
                in VersionSpec          version,
                in IDLType              result,
                in OperationMode        mode,
                in ParDescriptionSeq    params,
```

```
                in ExceptionDefSeq    exceptions,
                in ContextIdSeq       contexts
            );
        };

        struct InterfaceDescription {
            Identifier          name;
            RepositoryId        id;
            RepositoryId        defined_in;
            VersionSpec         version;
            RepositoryIdSeq     base_interfaces;
        };
    };
```

Read Interface

The **base_interfaces** attribute lists all the interfaces from which this interface inherits. The **is_a** operation returns TRUE if the interface on which it is invoked either is identical to or inherits, directly or indirectly, from the interface identified by its **interface_id** parameter. Otherwise it returns FALSE.

The **describe_interface** operation returns a **FullInterfaceDescription** describing the interface, including its operations and attributes.

The inherited **describe** operation for an **InterfaceDef** returns an **InterfaceDescription**.

The inherited **contents** operation returns the list of constants, typedefs, and exceptions defined in this InterfaceDef and the list of attributes and operations either defined or inherited in this InterfaceDef. If the **exclude_inherited** parameter is set to TRUE, only attributes and operations defined within this interface are returned. If the **exclude_inherited** parameter is set to FALSE, all attributes and operations are returned.

Write Interface

Setting the **base_interfaces** attribute returns an error if the **name** attribute of any object contained by this **InterfaceDef** conflicts with the **name** attribute of any object contained by any of the specified base **InterfaceDef**s.

The **create_attribute** operation returns a new **AttributeDef** contained in the **InterfaceDef** on which it is invoked. The **id**, **name**, **version, type_def**, and **mode** attributes are set as specified. The **type** attribute is also set. The **defined_in** attribute is initialized to identify the containing **InterfaceDef**. An error is returned if an object with the specified **id** already exists within this object's **Repository**, or if an object with the specified **name** already exists within this **InterfaceDef**.

The **create_operation** operation returns a new **OperationDef** contained in the **InterfaceDef** on which it is invoked. The **id**, **name**, **version**, **result_def**, **mode**, **params**, **exceptions**, and **contexts** attributes are set as specified. The **result** attribute is also set. The **defined_in** attribute is initialized to identify the containing

InterfaceDef. An error is returned if an object with the specified **id** already exists within this object's **Repository**, or if an object with the specified **name** already exists within this **InterfaceDef**.

8.6 RepositoryIds

RepositoryIds are values that can be used to establish the identity of information in the repository. A **RepositoryId** is represented as a string, allowing programs to store, copy, and compare them without regard to the structure of the value. It does not matter what format is used for any particular **RepositoryId**. However, conventions are used to manage the name space created by these IDs.

RepositoryIds may be associated with OMG IDL definitions in a variety of ways. Installation tools might generate them, they might be defined with pragmas in OMG IDL source, or they might be supplied with the package to be installed.

The format of the id is a short format name followed by a colon (":") followed by characters according to the format. This specification defines three formats: one derived from OMG IDL names, one that uses DCE UUIDs, and another intended for short-term use, such as in a development environment.

8.6.1 OMG IDL Format

The OMG IDL format for **RepositoryIds** primarily uses OMG IDL scoped names to distinguish between definitions. It also includes an optional unique prefix, and major and minor version numbers.

The **RepositoryId** consists of three components, separated by colons, (":")

The first component is the format name, "IDL."

The second component is a list of identifiers, separated by "/" characters. These identifiers are arbitrarily long sequences of alphabetic, digit, underscore ("_"), hyphen ("-"), and period (".") characters. Typically, the first identifier is a unique prefix, and the rest are the OMG IDL Identifiers that make up the scoped name of the definition.

The third component is made up of major and minor version numbers, in decimal format, separated by a ".". When two interfaces have **RepositoryId**s differing only in minor version number it can be assumed that the definition with the higher version number is upwardly compatible with (i.e., can be treated as derived from) the one with the lower minor version number.

8.6.2 DCE UUID Format

DCE UUID format **RepositoryId**s start with the characters "DCE:" and are followed by the printable form of the UUID, a colon, and a decimal minor version number, for example: "DCE:700dc518-0110-11ce-ac8f-0800090b5d3e:1".

8.6.3 LOCAL Format

Local format **RepositoryId**s start with the characters “LOCAL:” and are followed by an arbitrary string. Local format IDs are not intended for use outside a particular repository, and thus do not need to conform to any particular convention. Local IDs that are just consecutive integers might be used within a development environment to have a very cheap way to manufacture the IDs while avoiding conflicts with well-known interfaces.

8.6.4 Pragma Directives for RepositoryId

Three pragma directives (id, prefix, and version), are specified to accommodate arbitrary **RepositoryId** formats and still support the OMG IDL **RepositoryId** format with minimal annotation. The pragma directives can be used with the OMG IDL, DCE UUID, and LOCAL formats. An IDL compiler must either interpret these annotations as specified, or ignore them completely.

The ID Pragma

An OMG IDL pragma of the format

#pragma ID <name> “<id>”

associates an arbitrary **RepositoryId** string with a specific OMG IDL name. The **<name>** can be a fully or partially scoped name or a simple identifier, interpreted according to the usual OMG IDL name lookup rules relative to the scope within which the pragma is contained.

The Prefix Pragma

An OMG IDL pragma of the format:

#pragma prefix “<string>”

sets the current prefix used in generating OMG IDL format **RepositoryId**s. The specified prefix applies to **RepositoryId**s generated after the pragma until the end of the current scope is reached or another prefix pragma is encountered.

For example, the **RepositoryId** for the initial version of interface **Printer** defined on module **Office** by an organization known as “SoftCo” might be “IDL:SoftCo/Office/Printer:1.0”.

This format makes it convenient to generate and manage a set of IDs for a collection of OMG IDL definitions. The person creating the definitions sets a prefix (“SoftCo”), and the IDL compiler or other tool can synthesize all the needed IDs.

Because **RepositoryId**s may be used in many different computing environments and ORBs, as well as over a long period of time, care must be taken in choosing them. Prefixes that are distinct, such as trademarked names, domain names, UUIDs, and so forth, are preferable to generic names such as “document.”

The Version Pragma

An OMG IDL pragma of the format:

#pragma version <name> <major>.<minor>

provides the version specification used in generating an OMG IDL format **RepositoryId** for a specific OMG IDL name. The **<name>** can be a fully or partially scoped name or a simple identifier, interpreted according to the usual OMG IDL name lookup rules relative to the scope within which the pragma is contained. The **<major>** and **<minor>** components are decimal unsigned shorts.

If no version pragma is supplied for a definition, version 1.0 is assumed.

Generation of OMG IDL - Format IDs

A definition is globally identified by an OMG IDL - format **RepositoryId** if no ID pragma is encountered for it.

The ID string can be generated by starting with the string “IDL:”. Then, if any prefix pragma applies, it is appended, followed by a “/” character. Next, the components of the scoped name of the definition, relative to the scope in which any prefix that applies was encountered, are appended, separated by “/” characters. Finally, a “:” and the version specification are appended.

For example, the following OMG IDL:

```
module M1 {
    typedef long T1;
    typedef long T2;
    #pragma ID T2 “DCE:d62207a2-011e-11ce-88b4-0800090b5d3e:3”

};

#pragma prefix “P1”



module M2 {
    module M3 {
        #pragma prefix “P2”
        typedef long T3;
};
        typedef long T4;
#pragma version T4 2.4
};
```

specifies types with the following scoped names and **RepositoryId**s:

::M1::T1 IDL:M1/T1:1.0

::M1::T2 DCE:d62207a2-011e-11ce-88b4-0800090b5d3e:3

::M2::M3::T3 IDL:P2/T3:1.0

::M2::T4 IDL:P1/M2/T4:2.4

For this scheme to provide reliable global identity, the prefixes used must be unique. Two non-colliding options are suggested: Internet domain names and DCE UUIDs.

Furthermore, in a distributed world, where different entities independently evolve types, a convention must be followed to avoid the same **RepositoryId** being used for two different types. Only the entity that created the prefix has authority to create new IDs by simply incrementing the version number. Other entities must use a new prefix, even if they are only making a minor change to an existing type.

Prefix pragmas can be used to preserve the existing IDs when a module or other container is renamed or moved.

```
module M4 {
    #pragma prefix “P1/M2”
module M3 {
    #pragma prefix “P2”
        typedef long T3;
            };
        typedef long T4;
        #pragma version T4 2.4
};
```

This OMG IDL declares types with the same global identities as those declared in module M2 above.

8.6.5 For More Information

Section 8.8, “OMG IDL for Interface Repository,” on page 8-44 shows the OMG IDL specification of the IR, including the #pragma directive. “Preprocessing” on page 3-9 contains additional, general information on the pragma directive.

8.6.6 RepositoryIDs for OMG-Specified Types

Interoperability between implementations of official OMG specifications, including but not limited to CORBA, CORBAservices, and CORBAfacilities, depends on unambiguous specification of **RepositoryId**s for all IDL-defined types in such specifications. Unless **pragma** directives establishing **RepositoryId**s for all definitions are present in an IDL definition officially published by the OMG, the following directive is implicitly present at file scope preceding all such definitions:

#pragma prefix “omg.org”

For example, if an existing official specification included the IDL fragment:

```
module CORBA { // non-normative example IDL
    interface Nothing {
        void do_nothing();
    };
};
```

the **RepositoryId** of the interface would be

“IDL:omg.org/CORBA/Nothing:1.0”.

Revisions to OMG specifications must also ensure that the definitions associated with existing **RepositoryId**s are not changed. A **pragma version** or **pragma id** directive should be included with any revised IDL definition to specify a distinct identity for the revised type. If the revised definition is compatible with the previous definition, such as when a new operation is added to an existing interface, only the minor version should be incremented.

A revision of the previous example might look something like:

```
module CORBA { // revised non-normative example IDL
    interface Nothing {
        void do_nothing();
        void do_something();
    };
    #pragma version Nothing 1.1
};
```

for which the **RepositoryId** of the interface would be

“IDL:omg.org/CORBA/Nothing:1.1”.

If an implementation must extend an OMG-specified interface, interoperability requires it to derive a new interface from the standard interface, rather than modify the standard definition.

8.7 TypeCodes

TypeCodes are values that represent invocation argument types and attribute types. They can be obtained from the Interface Repository or from IDL compilers.

TypeCodes have a number of uses. They are used in the dynamic invocation interface to indicate the types of the actual arguments. They are used by an Interface Repository to represent the type specifications that are part of many OMG IDL declarations. Finally, they are crucial to the semantics of the **any** type.

TypeCodes are themselves values that can be passed as invocation arguments. To allow different ORB implementations to hide extra information in **TypeCode**s, the representation of **TypeCode**s will be opaque (like object references). However, we will assume that the representation is such that **TypeCode** "literals" can be placed in C include files.

Abstractly, **TypeCode**s consist of a "kind" field, and a set of parameters appropriate for that kind. For example, the **TypeCode** describing OMG IDL type **long** has kind **tk_long** and no parameters. The **TypeCode** describing OMG IDL type **sequence<boolean,10>** has kind **tk_sequence** and two parameters: **10** and **boolean**.

8.7.1 The TypeCode Interface

The PIDL interface for **TypeCodes** is as follows:

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

        interface TypeCode {
            exception       Bounds {};
            exception       BadKind {};

        // for all TypeCode kinds
        boolean         equal (in TypeCode tc);
        TCKind          kind ();
        // for tk_objref, tk_struct, tk_union, tk_enum, tk_alias, and
        tk_except
        RepositoryId    id () raises (BadKind);

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

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

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

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


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

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

        // for tk_fixed
        unsigned short fixed_digits() raises(BadKind);
        short fixed_scale() raises(BadKind);

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

With the above operations, any **TypeCode** can be decomposed into its constituent parts. The **BadKind** exception is raised if an operation is not appropriate for the **TypeCode** kind it invoked.

The **equal** operation can be invoked on any **TypeCode**. Equal **TypeCode**s are interchangeable, and give identical results when **TypeCode** operations are applied to them.

The **kind** operation can be invoked on any **TypeCode**. Its result determines what other operations can be invoked on the **TypeCode**.

The **id** operation can be invoked on object reference, structure, union, enumeration, alias, and exception **TypeCode**s. It returns the **RepositoryId** globally identifying the type. Object reference and exception **TypeCode**s always have a **RepositoryId**. Structure, union, enumeration, and alias **TypeCode**s obtained from the Interface Repository or the **ORB::create_operation_list** operation also always have a **RepositoryId**. Otherwise, the **id** operation can return an empty string.

The **name** operation can also be invoked on object reference, structure, union, enumeration, alias, and exception **TypeCode**s. It returns the simple name identifying the type within its enclosing scope. Since names are local to a **Repository**, the name returned from a **TypeCode** may not match the name of the type in any particular **Repository**, and may even be an empty string.

The **member_count** and **member_name** operations can be invoked on structure, union, and enumeration **TypeCode**s. **Member_count** returns the number of members constituting the type. **Member_name** returns the simple name of the member identified by **index**. Since names are local to a **Repository**, the name returned from a **TypeCode** may not match the name of the member in any particular **Repository**, and may even be an empty string.

The **member_type** operation can be invoked on structure and union **TypeCode**s. It returns the **TypeCode** describing the type of the member identified by **index**.

The **member_label**, **discriminator_type**, and **default_index** operations can only be invoked on union **TypeCode**s. **Member_label** returns the label of the union member identified by **index**. For the default member, the label is the zero octet. The **discriminator_type** operation returns the type of all non-default member labels. The **default_index** operation returns the index of the default member, or -1 if there is no default member.

The **member_name**, **member_type**, and **member_label** operations raise **Bounds** if the index parameter is greater than or equal to the number of members constituting the type.

The **content_type** operation can be invoked on sequence, array, and alias **TypeCode**s. For sequences and arrays, it returns the element type. For aliases, it returns the original type.

An array **TypeCode** only describes a single dimension of an OMG IDL array. Multi-dimensional arrays are represented by nesting **TypeCode**s, one per dimension. The outermost **tk_array Typecode** describes the leftmost array index of the array as defined in IDL. Its **content_type** describes the next index. The innermost nested **tk_array TypeCode** describes the rightmost index and the array element type.

The **length** operation can be invoked on string, wide string, sequence, and array **TypeCode**s. For strings and sequences, it returns the bound, with zero indicating an unbounded string or sequence. For arrays, it returns the number of elements in the array. For wide strings, it returns the bound, or zero for unbounded wide strings.

The deprecated **param_count** and **parameter** operations provide access to those parameters that were present in previous versions of *CORBA*. Some information available via other **TypeCode** operations is not visible via the **parameter** operation. The meaning of the indexed parameters for each **TypeCode** kind are listed in Table 8-1, along with the information that is not visible via the **parameter** operation.

Table 8-1 Legal TypeCode Kinds and Parameters

KIND	PARAMETER LIST	NOT VISIBLE
tk_null	*NONE*	
tk_void	*NONE*	
tk_short	*NONE*	
tk_long	*NONE*	
tk_longlong	*NONE*	
tk_ushort	*NONE*	
tk_ulong	*NONE*	
tk_ulonglong	*NONE*	
tk_float	*NONE*	
tk_double	*NONE*	
tk_longdouble	*NONE*	
tx_fixed	{digits_integer, scale_integer}	
tk_boolean	*NONE*	
tk_char	*NONE*	
tk_wchar	*NONE*	
tk_octet	*NONE*	
tk_any	*NONE*	
tk_TypeCode	*NONE*	
tk_Principal	*NONE*	
tk_objref	{ interface-id }	interface name
tk_struct	{ struct-name, member-name, TypeCode, ... (repeat pairs) }	RepositoryId
tk_union	{ union-name, discriminator-TypeCode, label-value, member-name, TypeCode, ... (repeat triples) }	RepositoryId
tk_enum	{ enum-name, enumerator-name, ... }	RepositoryId
tk_string	{ maxlen-integer }	
tk_wstring	{maxlen-integer}	
tk_sequence	{ TypeCode, maxlen-integer }	
tk_array	{ TypeCode, length-integer }	
tk_alias	{ alias-name, TypeCode }	Repositoryid
tk_except	{ except-name, member-name, TypeCode, ... (repeat pairs) }	RepositoryId

The **tk_fixed TypeCode** has 2 parameters: a non-zero integer specifying the precision of the fixed-point number in decimal digits, and an integer giving the position of the decimal point (scale).

The **tk_objref TypeCode** represents an interface type. Its parameter is the **RepositoryId** of that interface.

A structure with N members results in a **tk_struct TypeCode** with 2N+1 parameters: first, the simple name of the struct; the rest are member names alternating with the corresponding member **TypeCode**. Member names are represented as strings.

A union with N members results in a **tk_union TypeCode** with 3N+2 parameters: the simple name of the union, the discriminator **TypeCode** followed by a label value, member name, and member **TypeCode** for each of the N members. The label values are all values of the data type designated by the discriminator **TypeCode**, with one exception. The default member (if present) is marked with a label value consisting of the 0 **octet**. Recall that the operation "parameter(tc,i)" returns an **any**, and that anys themselves carry a **TypeCode** that can distinguish an octet from any of the legal switch types.

The **tk_enum TypeCode** has the simple name of the enum followed by the enumerator names as parameters. Enumerator names are represented as strings.

The **tk_string TypeCode** has one parameter: an integer giving the maximum string length. A maximum of 0 denotes an unbounded string.

The **tk_wstring TypeCode** has one parameter, an integer specifying the maximum length. A length of zero indicates an unbounded wide string.

The **tk_sequence TypeCode** has 2 parameters: a **TypeCode** for the sequence elements, and an integer giving the maximum sequence. Again, 0 denotes unbounded.

The **tk_array TypeCode** has 2 parameters: a **TypeCode** for the array elements, and an integer giving the array length. Arrays are never unbounded.

The **tk_alias TypeCode** has 2 parameters: the name of the alias followed by the **TypeCode** of the type being aliased.

The **tk_except TypeCode** has the same format as the **tk_struct TypeCode**, except that exceptions with no members are allowed.

8.7.2 TypeCode Constants

If "**typedef ... FOO;**" is an IDL type declaration, the IDL compiler will (if asked) produce a declaration of a **TypeCode** constant named TC_FOO for the C language mapping. In the case of an unnamed, bounded string type used directly in an operation or attribute declaration, a **TypeCode** constant named TC_string_n, where n is the bound of the string is produced. (For example, "string<4> op1();" produces the constant "TC_string_4".) These constants can be used with the dynamic invocation interface, and any other routines that require **TypeCode**s.

The IDL compiler will generate fixed-point decimal **TypeCode**s on request, much as it does for bounded strings. Where an unnamed fixed type of the form **fixed<d,s>** is used directly in an operation or attribute declaration, a **TypeCode** constant named "**`TC_fixed_d_s`**" is generated. For example, a **fixed** type with 10 decimal digits and

a scale factor of 4, **fixed<10,4>**, produces the constant "**TC_fixed_10_4**." The sign of a negative scale factor is represented by the letter "n;" thus the IDL type **fixed<4,-6>** would produce "**TC_fixed_4_n6**."

The predefined **TypeCode** constants, named according to the C language mapping, are:

```
TC_null
TC_void
TC_short
TC_long
TC_longlong
TC_ushort
TC_ulong
TC_ulonglong
TC_float
TC_double
TC_longdouble
TC_boolean
TC_char
TC_wchar
TC_octet
TC_any
TC_TypeCode
TC_Principal
TC_Object = tk_objref { Object }
TC_string= tk_string { 0 } // unbounded
TC_wstring = tk_wstring{0}              // unbounded
TC_CORBA_NamedValue= tk_struct { ... }
TC_CORBA_InterfaceDescription= tk_struct { ... }
TC_CORBA_OperationDescription= tk_struct { ... }
TC_CORBA_AttributeDescription= tk_struct { ... }
TC_CORBA_ParameterDescription= tk_struct { ... }
TC_CORBA_ModuleDescription= tk_struct { ... }
TC_CORBA_ConstantDescription= tk_struct { ... }
TC_CORBA_ExceptionDescription= tk_struct { ... }
TC_CORBA_TypeDescription= tk_struct { ... }
TC_CORBA_InterfaceDef_FullInterfaceDescription= tk_struct { ... }
```

The exact form for **TypeCode** constants is language mapping, and possibly implementation, specific.

8.7.3 Creating TypeCodes

When creating type definition objects in an Interface Repository, types are specified in terms of object references, and the **TypeCode**s describing them are generated automatically.

In some situations, such as bridges between ORBs, **TypeCode**s need to be constructed outside of any Interface Repository. This can be done using operations on the **ORB** pseudo-object.

```
module CORBA {
    interface ORB {
        // other operations ...

        TypeCode  create_struct_tc (
                in RepositoryId         id,
                in Identifier           name,
                in StructMemberSeq      members
        );

        TypeCode  create_union_tc (
                in RepositoryId         id,
                in Identifier           name,
                in TypeCode             discriminator_type,
                in UnionMemberSeq       members
        );

        TypeCode  create_enum_tc (
                in RepositoryId         id,
                in Identifier           name,
                in EnumMemberSeq        members
        );

        TypeCode create_alias_tc (
                in RepositoryId         id,
                in Identifier           name,
                in TypeCode             original_type
        );

        TypeCode  create_exception_tc (
                in RepositoryId         id,
                in Identifier           name,
                in StructMemberSeq      members
        );

        TypeCode  create_interface_tc (
                in RepositoryId         id,
                in Identifier           name
        );

        TypeCode  create_string_tc (
                in unsigned long        bound
        );
```

```
        TypeCode create_wstring_tc (
                in unsigned long    bound
        );

        TypeCode create_fixed_tc (
                in unsigned short digits,
                in short scale
        );

        TypeCode  create_sequence_tc (
                in unsigned long        bound,
                in TypeCode             element_type
        );

        TypeCode  create_recursive_sequence_tc (
                in unsigned long        bound,
                in unsigned long        offset
        );

        TypeCode  create_array_tc (
                in unsigned long        length,
                in TypeCode             element_type
        );
    };
};
```

Most of these operations are similar to corresponding IR operations for creating type definitions. **TypeCode**s are used here instead of **IDLType** object references to refer to other types. In the **StructMember** and **UnionMember** structures, only the **type** is used, and the **type_def** should be set to nil.

The **create_recursive_sequence_tc** operation is used to create **TypeCode**s describing recursive sequences. The result of this operation is used in constructing other types, with the **offset** parameter determining which enclosing **TypeCode** describes the elements of this sequence. For instance, to construct a **TypeCode** for the following OMG IDL structure, the offset used when creating its sequence **TypeCode** would be one:

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

Operations to create primitive **TypeCode**s are not needed, since **TypeCode** constants for these are available.

8.8 OMG IDL for Interface Repository

This section contains the complete OMG IDL specification for the Interface Repository.

```
#pragma prefix “omg.org”

module CORBA {
 typedef string Identifier;
 typedef string ScopedName;
 typedef string RepositoryId;

enum DefinitionKind {
     dk_none, dk_all,
     dk_Attribute, dk_Constant, dk_Exception, dk_Interface,
     dk_Module, dk_Operation, dk_Typedef,
     dk_Alias, dk_Struct, dk_Union, dk_Enum,
     dk_Primitive, dk_String, dk_Sequence, dk_Array,
     dk_Repository,
     dk_Wstring, dk_Fixed
 };

interface IRObject {
     // read interface
     readonly attribute DefinitionKind def_kind;
     // write interface
     void destroy ();
 };


typedef string VersionSpec;

 interface Contained;
 interface Repository;
 interface Container;

 interface Contained : IRObject {
     // read/write interface

     attribute RepositoryId id;
     attribute Identifier name;
     attribute VersionSpec version;

     // read interface

     readonly attribute Container defined_in;
     readonly attribute ScopedName absolute_name;
     readonly attribute Repository containing_repository;

     struct Description {
```

```
        DefinitionKind kind;
        any value;
      };

      Description describe ();

      // write interface

      void move (
         in Container new_container,
         in Identifier new_name,
         in VersionSpec new_version
         );
};

interface ModuleDef;
interface ConstantDef;
interface IDLType;
interface StructDef;
interface UnionDef;
interface EnumDef;
interface AliasDef;
interface InterfaceDef;
typedef sequence <InterfaceDef> InterfaceDefSeq;

typedef sequence <Contained> ContainedSeq;
struct StructMember {
    Identifier name;
    TypeCode type;
    IDLType type_def;
};

typedef sequence <StructMember> StructMemberSeq;

struct UnionMember {
    Identifier name;
    any label;
    TypeCode type;
    IDLType type_def;
};

typedef sequence <UnionMember> UnionMemberSeq;

typedef sequence <Identifier> EnumMemberSeq;

interface Container : IRObject {
    // read interface

    Contained lookup ( in ScopedName search_name);

    ContainedSeq contents (
```

```
    in DefinitionKind limit_type,
    in boolean exclude_inherited
    );

ContainedSeq lookup_name (
    in Identifier search_name,
    in long levels_to_search,
    in DefinitionKind limit_type,
    in boolean exclude_inherited
    );

struct Description {
    Contained contained_object;
    DefinitionKind kind;
    any value;
};

typedef sequence<Description> DescriptionSeq;

DescriptionSeq describe_contents (
    in DefinitionKind limit_type,
    in boolean exclude_inherited,
    in long max_returned_objs
    );


// write interface

ModuleDef create_module (
    in RepositoryId id,
    in Identifier name,
    in VersionSpec version
    );

ConstantDef create_constant (
    in RepositoryId id,
    in Identifier name,
    in VersionSpec version,
    in IDLType type,
    in any value
    );

StructDef create_struct (
    in RepositoryId id,
    in Identifier name,
    in VersionSpec version,
    in StructMemberSeq members
    );

UnionDef create_union (
    in RepositoryId id,
```

```
        in Identifier name,
        in VersionSpec version,
        in IDLType discriminator_type,
        in UnionMemberSeq members
        );

    EnumDef create_enum (
        in RepositoryId id,
        in Identifier name,
        in VersionSpec version,
        in EnumMemberSeq members
        );

    AliasDef create_alias (
        in RepositoryId id,
        in Identifier name,
        in VersionSpec version,
        in IDLType original_type
        );

    InterfaceDef create_interface (
        in RepositoryId id,
        in Identifier name,
        in VersionSpec version,
        in InterfaceDefSeq base_interfaces
        );

    ExceptionDef create_exception(
          in RepositoryId id,
          in Identifier name,
          in VersionSpec version,
          in StructMemberSeq members
    );
};


interface IDLType : IRObject {
    readonly attribute TypeCode type;
};


interface PrimitiveDef;
interface StringDef;
interface SequenceDef;
interface ArrayDef;

enum PrimitiveKind {
    pk_null, pk_void, pk_short, pk_long, pk_ushort, pk_ulong,
    pk_float, pk_double, pk_boolean, pk_char, pk_octet,
    pk_any, pk_TypeCode, pk_Principal, pk_string, pk_objref,
```

```
    pk_longlong, pk_ulonglong, pk_longdouble, pk_wchar, pk_wstring
};

interface Repository : Container {
    // read interface

    Contained lookup_id (in RepositoryId search_id);

    PrimitiveDef get_primitive (in PrimitiveKind kind);

    // write interface

    StringDef create_string (in unsigned long bound);

    WstringDef create_wstring (in unsigned long bound);

    SequenceDef create_sequence (
     in unsigned long bound,
     in IDLType element_type
     );

    ArrayDef create_array (
       in unsigned long length,
       in IDLType element_type
       );
};

    FixedDef create_fixed (
       in unsigned short digits,
       in short scale
       );
};

interface ModuleDef : Container, Contained {
};

struct ModuleDescription {
    Identifier name;
    RepositoryId id;
    RepositoryId defined_in;
    VersionSpec version;
};


interface ConstantDef : Contained {
    readonly attribute TypeCode type;
    attribute IDLType type_def;
    attribute any value;
};

struct ConstantDescription {
```

```
    Identifier name;
    RepositoryId id;
    RepositoryId defined_in;
    VersionSpec version;
    TypeCode type;
    any value;
};

interface TypedefDef : Contained, IDLType {
};

struct TypeDescription {
    Identifier name;
    RepositoryId id;
    RepositoryId defined_in;
    VersionSpec version;
    TypeCode type;
};

interface StructDef : TypedefDef, Container {
    attribute StructMemberSeq members;
};

interface UnionDef : TypedefDef, Container {
    readonly attribute TypeCode discriminator_type;
    attribute IDLType discriminator_type_def;
    attribute UnionMemberSeq members;
};

interface EnumDef : TypedefDef {
    attribute EnumMemberSeq members;
};

interface AliasDef : TypedefDef {
    attribute IDLType original_type_def;
};

interface PrimitiveDef: IDLType {
    readonly attribute PrimitiveKind kind;
};

interface StringDef : IDLType {
    attribute unsigned long bound;
```

```
};

interface WstringDef : IDLType {
    attribute unsigned long bound;
};

interface FixedDef : IDLType {
    attribute unsigned short digits;
    attribute short scale;
};

interface SequenceDef : IDLType {
    attribute unsigned long bound;
    readonly attribute TypeCode element_type;
    attribute IDLType element_type_def;
};


interface ArrayDef : IDLType {
    attribute unsigned long length;
    readonly attribute TypeCode element_type;
    attribute IDLType element_type_def;
};



interface ExceptionDef : Contained, Container {
    readonly attribute TypeCode type;
    attribute StructMemberSeq members;
};
struct ExceptionDescription {
    Identifier name;
    RepositoryId id;
    RepositoryId defined_in;
    VersionSpec version;
    TypeCode type;
};
enum AttributeMode {ATTR_NORMAL, ATTR_READONLY};

interface AttributeDef : Contained {
    readonly attribute TypeCode type;
    attribute IDLType type_def;
    attribute AttributeMode mode;
};

struct AttributeDescription {
    Identifier name;
    RepositoryId id;
    RepositoryId defined_in;
    VersionSpec version;
```

```
    TypeCode type;
    AttributeMode mode;
};

enum OperationMode {OP_NORMAL, OP_ONEWAY};

enum ParameterMode {PARAM_IN, PARAM_OUT, PARAM_INOUT};
struct ParameterDescription {
    Identifier name;
    TypeCode type;
    IDLType type_def;
    ParameterMode mode;
};
typedef sequence <ParameterDescription> ParDescriptionSeq;

typedef Identifier ContextIdentifier;
typedef sequence <ContextIdentifier> ContextIdSeq;

typedef sequence <ExceptionDef> ExceptionDefSeq;
typedef sequence <ExceptionDescription> ExcDescriptionSeq;

interface OperationDef : Contained {
    readonly attribute TypeCode result;
    attribute IDLType result_def;
    attribute ParDescriptionSeq params;
    attribute OperationMode mode;
    attribute ContextIdSeq contexts;
    attribute ExceptionDefSeq exceptions;
};

struct OperationDescription {
    Identifier name;
    RepositoryId id;
    RepositoryId defined_in;
    VersionSpec version;
    TypeCode result;
    OperationMode mode;
    ContextIdSeq contexts;
    ParDescriptionSeq parameters;
    ExcDescriptionSeq exceptions;
};

typedef sequence <RepositoryId> RepositoryIdSeq;
typedef sequence <OperationDescription> OpDescriptionSeq;
typedef sequence <AttributeDescription> AttrDescriptionSeq;

interface InterfaceDef : Container, Contained, IDLType {
    // read/write interface
```

```
        attribute InterfaceDefSeq base_interfaces;

        // read interface

        boolean is_a (in RepositoryId interface_id);

        struct FullInterfaceDescription {
         Identifier name;
         RepositoryId id;
         RepositoryId defined_in;
         VersionSpec version;
         OpDescriptionSeq operations;
         AttrDescriptionSeq attributes;
         RepositoryIdSeq base_interfaces;
         TypeCode type;
        };

        FullInterfaceDescription describe_interface();

        // write interface

        AttributeDef create_attribute (
         in RepositoryId id,
         in Identifier name,
         in VersionSpec version,
         in IDLType type,
         in AttributeMode mode
         );

        OperationDef create_operation (
         in RepositoryId id,
         in Identifier name,
         in VersionSpec version,
         in IDLType result,
         in OperationMode mode,
         in ParDescriptionSeq params,
         in ExceptionDefSeq exceptions,
         in ContextIdSeq contexts
         );
    };

    struct InterfaceDescription {
        Identifier name;
        RepositoryId id;
        RepositoryId defined_in;
        VersionSpec version;
        RepositoryIdSeq base_interfaces;
    };


    enum TCKind {
```

```
    tk_null, tk_void,
    tk_short, tk_long, tk_ushort, tk_ulong,
    tk_float, tk_double, tk_boolean, tk_char,
    tk_octet, tk_any, tk_TypeCode, tk_Principal, tk_objref,
    tk_struct, tk_union, tk_enum, tk_string,
    tk_sequence, tk_array, tk_alias, tk_except
    tk_longlong, tk_ulonglong, tk_longdouble,
    tk_wchar, tk_wstring, tk_fixed
};

interface TypeCode { // PIDL
    exception Bounds {};
    exception BadKind {};

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

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

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

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

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

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

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

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

    // for tk_fixed
    unsigned short fixed_digits() raises (BadKind);
    short fixed_scale() raises (BadKind);

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

```
};

interface ORB {
    // other operations ...

    TypeCode create_struct_tc (
        in RepositoryId id,
        in Identifier name,
        in StructMemberSeq members
        );

    TypeCode create_union_tc (
        in RepositoryId id,
        in Identifier name,
        in TypeCode discriminator_type,
        in UnionMemberSeq members
        );

    TypeCode create_enum_tc (
        in RepositoryId id,
        in Identifier name,
        in EnumMemberSeq members
        );

    TypeCode create_alias_tc (
        in RepositoryId id,
        in Identifier name,
        in TypeCode original_type
        );

    TypeCode create_exception_tc (
        in RepositoryId id,
        in Identifier name,
        in StructMemberSeq members
        );

    TypeCode create_interface_tc (
        in RepositoryId id,
        in Identifier name
        );

    TypeCode create_string_tc (
        in unsigned long bound
        );

    TypeCode create_wstring_tc (
        in unsigned long bound
        );

    TypeCode create_fixed_tc (
        in unsigned short digits,
```

```
            in short scale
            );

        TypeCode create_sequence_tc (
            in unsigned long bound,
            in TypeCode element type
            );

        TypeCode create_recursive_sequence_tc (
            in unsigned long bound,
            in unsigned long offset
            );

        TypeCode create_array_tc (
            in unsigned long length,
            in TypeCode element_type
            );
    };
};
```