Naming Service Specification

October 2004
Version 1.3
formal/04-10-03

OBJECT MANAGEMENT GROUP

An Available Specification of the Object Management Group, Inc.

OMG’s Issue Reporting Procedure

All OMG specifications are subject to continuous review and improvement. As part of this process we encourage readers to report any ambiguities, inconsistencies, or inaccuracies they may find by completing the Issue Reporting Form listed on the main web page *http://www.omg.org*, under Documents, Report a Bug/Issue (http://www.omg.org/technology/agreement.htm).

Contents

Contents

Preface

About This Document

Under the terms of the collaboration between OMG and The Open Group, this document is a candidate for adoption by The Open Group, as an Open Group Technical Standard. The collaboration between OMG and The Open Group ensures joint review and cohesive support for emerging object-based specifications.

Object Management Group

The Object Management Group, Inc. (OMG) is an international organization supported by over 600 members, including information system vendors, software developers and users. Founded in 1989, the OMG promotes the theory and practice of object-oriented technology in software development. The organization's charter includes the establishment of industry guidelines and object management specifications to provide a common framework for application development. Primary goals are the reusability, portability, and interoperability of object-based software in distributed, heterogeneous environments. Conformance to these specifications will make it possible to develop a heterogeneous applications environment across all major hardware platforms and operating systems.

OMG's objectives are to foster the growth of object technology and influence its direction by establishing the Object Management Architecture (OMA). The OMA provides the conceptual infrastructure upon which all OMG specifications are based. More information is available at http://www.omg.org/.

Associated OMG Documents

The CORBA documentation is organized as follows:

- *Object Management Architecture Guide* defines the OMG's technical objectives and terminology and describes the conceptual models upon which OMG standards are based. It defines the umbrella architecture for the OMG standards. It also provides information about the policies and procedures of OMG, such as how standards are proposed, evaluated, and accepted.
- CORBA Platform Technologies
  - *CORBA: Common Object Request Broker Architecture and Specification* contains the architecture and specifications for the Object Request Broker.
  - *CORBA Languages*, a collection of language mapping specifications. See the individual language mapping specifications.
  - *CORBA Services*, a collection of specifications for OMG's Object Services. See the individual service specifications.
  - *CORBA Facilities*, a collection of specifications for OMG's Common Facilities. See the individual facility specifications.
- CORBA Domain Technologies
  - *CORBA Manufacturing*, a collection of specifications that relate to the manufacturing industry. This group of specifications defines standardized object-oriented interfaces between related services and functions.
  - *CORBA Med*, a collection of specifications that relate to the healthcare industry and represents vendors, healthcare providers, payers, and end users.
  - *CORBA Finance*, a collection of specifications that target a vitally important vertical market: financial services and accounting. These important application areas are present in virtually all organizations: including all forms of monetary transactions, payroll, billing, and so forth.
  - *CORBA Telecoms*, a collection of specifications that relate to the OMG-compliant interfaces for telecommunication systems.

The OMG collects information for each book in the documentation set by issuing Requests for Information, Requests for Proposals, and Requests for Comment and, with its membership, evaluating the responses. Specifications are adopted as standards only when representatives of the OMG membership accept them as such by vote. (The policies and procedures of the OMG are described in detail in the *Object Management Architecture Guide*.)

You may contact the Object Management Group, Inc. at:

OMG Headquarters
250 First Avenue
Needham, MA 02494
USA
Tel: +1-781-444-0404
Fax: +1-781-444-0320
pubs@omg.org
http://www.omg.org

Acknowledgments

The following companies submitted and/or supported parts of the *Interoperable Naming Service* specification:

- BEA Systems
- DSTC
- Inprise
- IONA Technologies, Ltd.

Service Description 1

Note – "Interoperable Naming Service" will be referred to as "Naming Service" throughout this specification.

Contents

This chapter contains the following topics.

Topic	**Page**
"Overview"	1-2
"Names"	1-2
"Example Scenarios"	1-3
"Design Principles"	1-4

Source Document(s)

This formal specification is based on the following OMG document(s):

- formal/00-11-01 - version 1.0
- formal/01-02-65 - version 1.1
- Editorial issue # 4246
- Lightweight Services specification - formal/04-10-01

1.1 Overview

A name-to-object association is called a *name binding*. A name binding is always defined relative to a *naming context*. A naming context is an object that contains a set of name bindings in which each name is unique. Different names can be bound to an object in the same or different contexts at the same time. There is no requirement, however, that all objects must be named.

To *resolve a name* is to determine the object associated with the name in a given context. To *bind a name* is to create a name binding in a given context. A name is always resolved relative to a context — there are no absolute names.

Because a context is like any other object, it can also be bound to a name in a naming context. Binding contexts in other contexts creates a *naming graph* — a directed graph with nodes and labeled edges where the nodes are contexts. A naming graph allows more complex names to reference an object. Given a context in a naming graph, a sequence of names can reference an object. This sequence of names (called a *compound name*) defines a path in the naming graph to navigate the resolution process. Figure 1-1 shows an example of a naming graph.

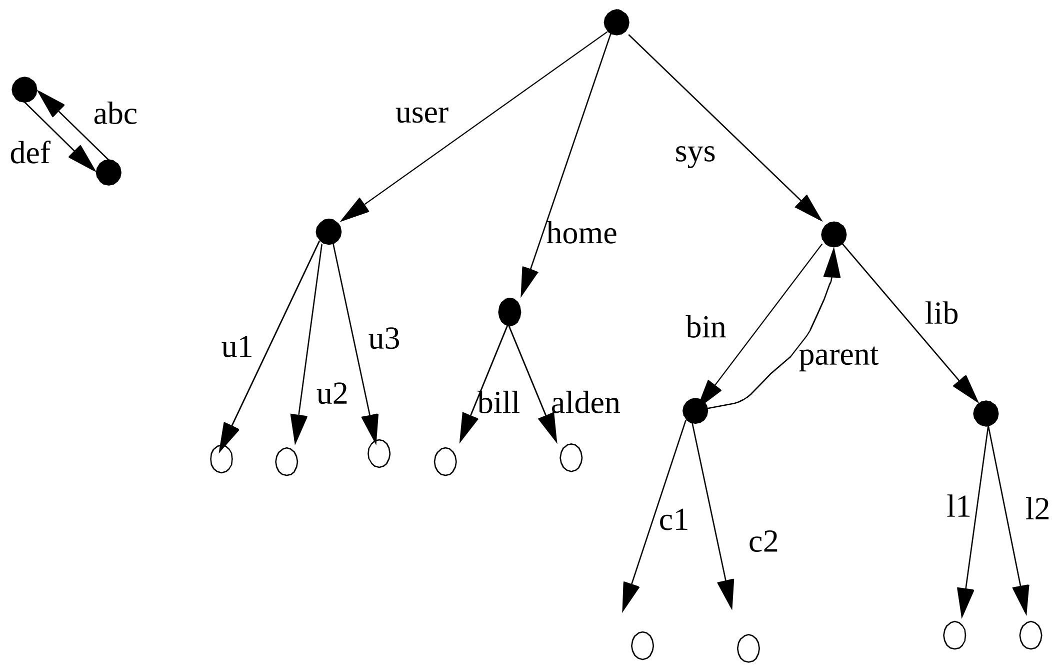


Figure 1-1 A Naming Graph

1.2 Names

Many of the operations defined on a naming context take names as parameters. Names have structure. A name is an ordered sequence of *components*.

A name with a single component is called a *simple name*; a name with multiple components is called a *compound name*. Each component except the last is used to name a context; the last component denotes the bound object. The notation:

component1/component2/component3

indicates a sequence of components.

Note – The slash (/) characters are simply a notation used here and are not intended to imply that names are sequences of characters separated by slashes.

A name component consists of two attributes: the **id attribute** and the **kind attribute**. Both the **id** attribute and the **kind** attribute are represented as IDL strings.

The **kind** attribute adds descriptive power to names in a syntax-independent way. Examples of the value of the **kind** attribute include *c_source*, *object_code*, *executable*, *postscript*, or “ ”. The naming system does not interpret, assign, or manage these values in any way. Higher levels of software may make policies about the use and management of these values. This feature addresses the needs of applications that use syntactic naming conventions to distinguish related objects. For example Unix uses suffixes such as **.c** and **.o**. Applications (such as the C compiler) depend on these syntactic convention to make name transformations (for example, to transform **foo.c** to **foo.o**).

A sequence of **id** and **kind** pairs forming a name can be expressed as a single string using the syntax described in Section 2.3, “The BindingIterator Interface,” on page 2-11. This allows names to be written down easily or to be presented as a strings in user interfaces. In addition, Section 2.4, “Stringified Names,” on page 2-12 describes a way to express a name relative to a particular naming context in URL format. The URL representation provides a human-readable form of an object reference that is named in some naming context.

1.3 Example Scenarios

This section provides two short scenarios that illustrate how the naming service specification can be used by two fairly different kinds of systems -- systems that differ in the kind of implementations used to build the Naming Service and that differ in models of how clients might use the Naming Service with other object services to locate objects.

In one system, the Naming Service is implemented using an underlying enterprise-wide naming server such as DCE CDS. The Naming Service is used to construct large, enterprise-wide naming graphs where NamingContexts model “directories” or “folders” and other names identify “document” or “file” kinds of objects. In other words, the naming service is used as the backbone of an enterprise-wide filing system. In such a system, non-object-based access to the naming service may well be as commonplace as object-based access to the naming service.

The Naming Service provides the principal mechanism through which most clients of an ORB-based system locate objects that they intend to use (make requests of). Given an initial naming context, clients navigate naming contexts retrieving lists of the names bound to that context. In conjunction with properties and security services, clients look for objects with certain “externally visible” characteristics, for example, for objects with recognized names or objects with a certain time-last-modified (all subject to security considerations). All objects used in such a scheme register their externally visible characteristics with other services (a name service, a properties service, and so on).

Conventions are employed in such a scheme that meaningfully partition the name space. For example, individuals are assigned naming contexts for personal use, groups of individuals may be assigned shared naming contexts while other contexts are organized in a public section of the naming graph. Similarly, conventions are used to identify contexts that list the names of services that are available in the system (e.g., that locate a translation or printing service).

In an alternative system, the Naming Service can be used in a more limited role and can have a less sophisticated implementation. In this model, naming contexts represent the types and locations of services that are available in the system and a much shallower naming graph is employed. For example, the Naming Service is used to register the object references of a mail service, an information service, a filing service.

Given a handful of references to “root objects” obtained from the Naming Service, a client uses the Relationship and Query Services to locate objects contained in or managed by the services registered with the Naming Service. In such a system, the Naming Service is used sparingly and instead clients rely on other services such as query services to navigate through large collections of objects. Also, objects in this scheme rarely register “external characteristics” with another service - instead they support the interfaces of Query or Relationship Services.

Of course, nothing precludes the Naming Service presented here from being used to provide both models of use at the same time. These two scenarios demonstrate how this specification is suitable for use in two fairly different kinds of systems with potentially quite different kinds of implementations. The service provides a basic building block on which higher-level services impose the conventions and semantics which determine how frameworks of application and facilities objects locate other objects.

1.4 Design Principles

Several principles have driven the design of the Naming Service:

1. The design imparts no semantics or interpretation of the names themselves; this is up to higher-level software.
2. The design supports distributed, heterogeneous implementation and administration of names and name contexts.

3. Naming service clients need not be aware of the physical site of name servers in a distributed environment, or which server interprets what portion of a compound name, or of the way that servers are implemented.
4. The Naming Service is a fundamental object service, with no dependencies on other interfaces.
5. Name contexts of arbitrary and unknown implementation may be utilized together as nested graphs of nodes that cooperate in resolving names for a client. No “universal” root is needed for a name hierarchy.
6. Existing name and directory services employed in different network computing environments can be transparently encapsulated using name contexts. All of the above features contribute to making this possible.
7. The design does not address namespace administration. It is the responsibility of higher-level software to administer the namespace.

Modules and Interfaces 2

Note – “Interoperable Naming Service” will be referred to as “Naming Service” throughout this specification.

Contents

This chapter contains the following topics.

Topic	**Page**
“The CosNaming Module”	2-1
“NamingContext Interface”	2-4
“The BindingIterator Interface”	2-11
“Stringified Names”	2-12
“URL schemes”	2-14
“Initial Reference to a NamingContextExt”	2-18

2.1 The CosNaming Module

The **CosNaming** module is a collection of interfaces that together define the Naming Service. This module contains three interfaces:

- The **NamingContext** interface
- The **BindingIterator** interface
- The **NamingContextExt** interface

This section describes these interfaces and their operations in detail.

The **CosNaming** module is shown below.

Note – Istring was a “placeholder for a future IDL internationalized string data type” in the original specification. It is maintained solely for compatibility reasons.

```
// File: CosNaming.idl
#ifndef _COSNAMING_IDL_
#define _COSNAMING_IDL_

#pragma prefix "omg.org"

module CosNaming {
    typedef string Istring;

    struct NameComponent {
        Istring id;
        Istring kind;
    };
    typedef sequence<NameComponent> Name;

    enum BindingType { nobject, ncontext };

    struct Binding {
        Name            binding_name;
        BindingType     binding_type;
    };

    // Note: In struct Binding, binding_name is incorrectly defined
    // as a Name instead of a NameComponent. This definition is
    // unchanged for compatibility reasons.
    typedef sequence <Binding> BindingList;

    interface BindingIterator;

    interface NamingContext {
        enum NotFoundReason {
            missing_node, not_context, not_object
        };

        exception NotFound {
            NotFoundReason      why;
            Name                rest_of_name;
        };

        exception CannotProceed {
            NamingContext       cxt;
            Name                rest_of_name;
        };
        exception InvalidName{};

        exception AlreadyBound {};

        exception NotEmpty{};

        void        bind(in Name n, in Object obj)
            raises(
                NotFound, CannotProceed,
                InvalidName, AlreadyBound
            );

        void        rebind(in Name n, in Object obj)
            raises(NotFound, CannotProceed, InvalidName);

        void        bind_context(in Name n, in NamingContext nc)
            raises(NotFound, CannotProceed,
                 InvalidName, AlreadyBound
            );

        void        rebind_context(in Name n, in NamingContext nc)
            raises(NotFound, CannotProceed, InvalidName);

        Object      resolve (in Name n)
            raises(NotFound, CannotProceed, InvalidName);

        void        unbind(in Name n)
            raises(NotFound, CannotProceed, InvalidName);

        NamingContext       new_context();
        NamingContext       bind_new_context(in Name n)
            raises(
                NotFound, AlreadyBound,
                CannotProceed, InvalidName
        );

        void        destroy() raises(NotEmpty);

        void        list(
                in unsigned long        how_many,
                out BindingList         bl,
                out BindingIterator     bi
        );

    };

    interface BindingIterator {
        boolean next_one(out Binding b);
        boolean next_n(in unsigned long, how_many, out BindingList bl);
        void        destroy();
    };
```

```
        interface NamingContextExt: NamingContext {
            typedef string StringName;
            typedef string Address;
            typedef string URLString;

            StringName      to_string(in Name n) raises(InvalidName);
            Name            to_name(in StringName sn)
                                raises(InvalidName);

            exception InvalidAddress {};

            URLString       to_url(in Address addr, in StringName sn)
                                raises(InvalidAddress, InvalidName);

            Object          resolve_str(in StringName sn)
                                raises(
                                    NotFound, CannotProceed,
                                    InvalidName
                                    );
        };
    };
    #endif // _COSNAMING_IDL_
```

2.1.1 Resolution of Compound Names

In this specification operations that are performed on compound names recursively perform the *equivalent* of a **resolve** operation on all but the last component of a name before performing the operation on the final name component. The general form is defined as follows:

ctx->op(<c1; c2; ...; cn>) equiv

ctx->resolve(<c1>)->resolve(<c2; cn-1>)->op(<cn>)

where ctx is a naming context, <c1; ...; cn> a compound name, and op a naming context operation.

Note – The intermediate components, <c1: ...; cn> of the compound name must have been bound using **bind_context** or **rebind_context** to take part in the resolve.

2.2 NamingContext Interface

The following sections describe the naming context data types and interface in detail.

2.2.1 Structures

2.2.1.1 NameComponent

```
struct NameComponent {
    Istring Id;
    Istring kind;
};
```

A name component consists of two attributes: the identifier attribute - **id** and the kind attribute - **kind**.

Both of these attributes are arbitrary-length strings of ISO Latin-1 characters, excluding the ASCII **NUL** character.

When comparing two **NameComponents** for equality both the **id** and the **kind** field must match in order for two **NameComponents** to be considered identical. This applies for zero-length (empty) fields as well. Name comparisons are case sensitive.

An implementation may place limitations on the characters that may be contained in a name component, as well as the length of a name component. For example, an implementation may disallow certain characters, may not accept the empty string as a legal name component, or may limit name components to some maximum length.

2.2.1.2 Name

A name is a sequence of **NameComponent**s. The empty sequence is not a legal name. An implementation may limit the length of the sequence to some maximum. When comparing **Name**s for equality, each **NameComponent** in the first name must match the corresponding **NameComponent** in the second **Name** for the names to be considered identical.

2.2.1.3 Binding

```
enum BindingType { nobject, ncontext };
struct Binding {
    Name            binding_name;
    BindingType     binding_type;
};
typedef sequence<Binding> BindingList;
```

This type is used by the **NamingContext::list**, **BindingIterator::next_n**, and **BindingIterator::next_one** operations. A **Binding** contains a **Name** in the member **binding_name**, together with the **BindingType** of that **Name** in the member **binding_type**.

Note – The **binding_name** member is incorrectly typed as a **Name** instead of a **NameComponent**. For compatibility with the original **CosNaming** specification this incorrect definition has been retained. The **binding_name** is used as a **NameComponent** and will always be a **Name** with length of 1.

The value of **binding_type** is **ncontext** if a **Name** denotes a binding created with one of the following operations:

- **bind_context**
- **rebind_context**
- **bind_new_context**

For bindings created with any other operation, the value of **BindingType** is **nobject**.

2.2.2 Exceptions

The Naming Service exceptions are defined below.

2.2.2.1 NotFound

```
exception NotFound {
    NotFoundReason why;
    Name rest_of_name;
};
```

This exception is raised by operations when a component of a name does not identify a binding or the type of the binding is incorrect for the operation being performed. The **why** member explains the reason for the exception and the **rest_of_name** member contains the remainder of the non-working name:

- **missing_node**

  The first name component in **rest_of_name** denotes a binding that is not bound under that name within its parent context.
- **not_context**

  The first name component in **rest_of_name** denotes a binding with a type of **nobject** when the type **ncontext** was required.
- **not_object**

  The first name component in **rest_of_name** denotes a binding with a type of **ncontext** when the type **nobject** was required.

2.2.2.2 CannotProceed

```
exception CannotProceed {
    NamingContext cxt;
    Name rest_of_name;
};
```

This exception is raised when an implementation has given up for some reason. The client, however, may be able to continue the operation at the returned naming context.

The **cxt** member contains the context that the operation may be able to retry from.

The **rest_of_name** member contains the remainder of the non-working name.

2.2.2.3 InvalidName

```
exception InvalidName {};
```

This exception is raised if a **Name** is invalid. A name of length zero is invalid (containing no name components). Implementations may place further limitations on what constitutes a legal name and raise this exception to indicate a violation.

2.2.2.4 AlreadyBound

```
exception AlreadyBound {};
```

Indicates an object is already bound to the specified name. Only one object can be bound to a particular **Name** in a context.

2.2.2.5 NotEmpty

```
exception NotEmpty {};
```

This exception is raised by **destroy** if the **NamingContext** contains bindings. A **NamingContext** must be empty to be destroyed.

2.2.3 Binding Objects

The binding operations name an object in a naming context. Once an object is bound, it can be found with the **resolve** operation. The Naming Service supports four operations to create bindings: **bind**, **rebind**, **bind_context**, and **rebind_context**. **bind_new_context** also creates a binding, see Section 2.2.6, “Creating Naming Contexts,” on page 2-9.

```
void bind(in Name n, in Object obj)
            raises(NotFound, CannotProceed, InvalidName, AlreadyBound);
void rebind(in Name n, in Object obj)
            raises(NotFound, CannotProceed, InvalidName);
void bind_context(in Name n, in NamingContext nc)
            raises(NotFound, CannotProceed, InvalidName, AlreadyBound);
void rebind_context(in Name n, in NamingContext nc)
            raises(NotFound, CannotProceed, InvalidName);
```

2.2.3.1 bind

Creates an **nobject** binding in the naming context.

2.2.3.2 rebind

Creates an **nobject** binding in the naming context even if the name is already bound in the context.

If already bound, the previous binding must be of type **nobject**; otherwise, a NotFound exception with a why reason of **not_object** is raised.

2.2.3.3 bind_context

Creates an **ncontext** binding in the parent naming context. Attempts to bind a nil context raise a BAD_PARAM exception.

2.2.3.4 rebind_context

Creates an **ncontext** binding in the naming context even if the name is already bound in the context.

If already bound, the previous binding must be of type **ncontext**; otherwise, a NotFound exception with a **why** reason of **not_context** will be raised.

2.2.3.5 Usage

If a binding with the specified name already exists, **bind** and **bind_context** raise an AlreadyBound exception.

If an implementation places limits on the number of bindings within a context, **bind** and **bind_context** raise the IMP_LIMIT system exception if the new binding cannot be created.

Naming contexts bound using **bind_context** and **rebind_context** participate in name resolution when compound names are passed to be resolved; naming contexts bound with **bind** and **rebind** do not.

Use of **rebind_context** may leave a potential orphaned context (one that is unreachable within an instance of the Name Service). Policies and administration tools regarding potential orphan contexts are implementation-specific.

If **rebind** or **rebind_context** raise a NotFound exception because an already existing binding is of the wrong type, the **rest_of_name** member of the exception has a sequence length of 1.

2.2.4 Resolving Names

The **resolve** operation is the process of retrieving an object bound to a name in a given context. The given name must exactly match the bound name. The naming service does not return the type of the object. Clients are responsible for “narrowing” the object to the appropriate type. That is, clients typically cast the returned object from **Object** to a more specialized interface. The IDL definition of the **resolve** operation is:

```
Object resolve (in Name n)
                raises (NotFound, CannotProceed, InvalidName);
```

Names can have multiple components; therefore, name resolution can traverse multiple contexts. These contexts can be federated between different Naming Service instances.

2.2.5 Unbinding Names

The **unbind** operation removes a name binding from a context. The definition of the **unbind** operation is:

```
void unbind(in Name n)
            raises (NotFound, CannotProceed, InvalidName);
```

2.2.6 Creating Naming Contexts

The Naming Service supports two operations to create new contexts: **new_context** and **bind_new_context**.

```
NamingContext new_context();
NamingContext bind_new_context(in Name n)
            raises(NotFound, AlreadyBound, CannotProceed, InvalidName);
```

2.2.6.1 new_context

This operation returns a new naming context. The new context is not bound to any name.

2.2.6.2 bind_new_context

This operation creates a new context and creates an **ncontext** binding for it using the name supplied as an argument.

2.2.6.3 Usage

If an implementation places limits on the number of naming contexts, both **new_context** and **bind_new_context** can raise the IMP_LIMIT system exception if the context cannot be created. **bind_new_context** can also raise IMP_LIMIT if the bind would cause an implementation limit on the number of bindings in a context to be exceeded.

2.2.7 Deleting Contexts

The **destroy** operation deletes a naming context.

```
void destroy()
        raises(NotEmpty);
```

This operation destroys its naming context. If there are bindings denoting the destroyed context, these bindings are *not* removed. If the naming context contains bindings, the operation raises NotEmpty.

2.2.8 Listing a Naming Context

The **list** operation allows a client to iterate through a set of bindings in a naming context.

```
void list (in unsigned long how_many,
      out BindingList bl, out BindingIterator bi);
};
```

list returns the bindings contained in a context in the parameter **bl**. The **bl** parameter is a sequence where each element is a **Binding** containing a **Name** of length 1 representing a single **NameComponent**.

The **how_many** parameter determines the maximum number of bindings to return in the parameter **bl**, with any remaining bindings to be accessed through the returned **BindingIterator bi**.

- A non-zero value of **how_many** guarantees that **bl** contains at most **how_many** elements. The implementation is free to return fewer than the number of bindings requested by **how_many**. However, for a non-zero value of **how_many**, it may not return a **bl** sequence with zero elements unless the context contains no bindings.
- If **how_many** is set to zero, the client is requesting to use only the **BindingIterator bi** to access the bindings and **list** returns a zero length sequence in **bl**.
- The parameter **bi** returns a reference to an iterator object.
    - If the **bi** parameter returns a non-nil reference, this indicates that the call to **list** may not have returned all of the bindings in the context and that the remaining bindings (if any) must be retrieved using the iterator. This applies for all values of **how_many**.

- If the **bi** parameter returns a nil reference, this indicates that the **bl** parameter contains all of the bindings in the context. This applies for all values of **how_many**.

2.3 The BindingIterator Interface

The **BindingIterator** interface allows a client to iterate through the bindings using the **next_one** or **next_n** operations:

If a context is modified in between calls to **list**, **next_one**, or **next_n**, the behavior of further calls to **next_one** or **next_n** is implementation-dependent.

```
interface BindingIterator {
    boolean next_one(out Binding b);
    boolean next_n(in unsigned long how_many,
                    out BindingList bl);
    void destroy();
};
```

2.3.1 Operations

2.3.1.1 next_one

The **next_one** operation returns the next binding. It returns true if it is returning a binding, false if there are no more bindings to retrieve. If **next_one** returns false, the returned **Binding** is indeterminate.

Further calls to **next_one** after it has returned false have undefined behavior.

2.3.1.2 next_n

next_n returns, in the parameter **bl**, bindings not yet retrieved with **list** or previous calls to **next_n** or **next_one**. It returns true if **bl** is a non-zero length sequence; it returns false if there are no more bindings and **bl** is a zero-length sequence.

The **how_many** parameter determines the maximum number of bindings to return in the parameter **bl**:

- A non-zero value of **how_many** guarantees that **bl** contains at most **how_many** elements. The implementation is free to return fewer than the number of bindings requested by **how_many**. However, it may not return a **bl** sequence with zero elements unless there are no bindings to retrieve.
- A zero value of **how_many** is illegal and raises a BAD_PARAM system exception.

next_n returns false with a **bl** parameter of length zero once all bindings have been retrieved. Further calls to **next_n** after it has returned a zero-length sequence have undefined behavior.

2.3.1.3 destroy

The **destroy** operation destroys its iterator. If a client invokes any operation on an iterator after calling **destroy**, the operation raises OBJECT_NOT_EXIST.

2.3.2 Garbage Collection of Iterators

Clients that create iterators but never call **destroy** can cause an implementation to permanently run out of resources. To protect itself against this scenario, an implementation is free to destroy an iterator object at any time without warning, using whatever algorithm it considers appropriate to choose iterators for destruction. In order to be robust in the presence of garbage collection, clients should be written to expect OBJECT_NOT_EXIST from calls to an iterator and handle this exception gracefully.

2.4 Stringified Names

Names are sequences of name components. This representation makes it difficult for applications to conveniently deal with names for I/O purposes, human or otherwise. This specification defines a syntax for stringified names and provides operations to convert a name in stringified form to its equivalent sequence form and vice-versa (see Section 2.5.4, “Converting between CosNames, Stringified Names, and URLs,” on page 2-17).

A stringified name represents one and only one **CosNaming::Name**. If two names are equal, their stringified representations are equal (and vice-versa).

The stringified name representation reserves use of the characters ‘/’, ‘.’, and ‘\’. The forward slash ‘/’ is a name component separator; the dot ‘.’ separates **id** and **kind** fields. The backslash ‘\’ is an escape character (see Section 2.4.2, “Escape Mechanism,” on page 2-13).

2.4.1 Basic Representation of Stringified Names

A stringified name consists of the name components of a name separated by a ‘/’ character. For example, a name consisting of the components “a,” “b,” and “c” (in that order) is represented as

```
a/b/c
```

Stringified names use the ‘.’ character to separate **id** and **kind** fields in the stringified representation. For example, the stringified name

```
a.b/c.d/.
```

represents the **CosNaming::Name**:

Index	id	kind
0	**a**	**b**
1	**c**	**d**
2	**<empty>**	**<empty>**

The single '.' character is the only representation of a name component with empty **id** and **kind** fields.

If a name component in a stringified name does not contain a '.' character, the entire component is interpreted as the **id** field, and the **kind** field is empty. For example:

a/./c.d/.e

corresponds to the **CosNaming::Name**:

Index	id	kind
0	**a**	**<empty>**
1	**<empty>**	**<empty>**
2	**c**	**d**
3	**<empty>**	**e**

If a name component has a non-empty **id** field and an empty **kind** field, the stringified representation consists only of the **id** field. A trailing '.' character is not permitted.

2.4.2 Escape Mechanism

The backslash '\' character escapes the reserved meaning of '/', '.', and '\' in a stringified name. The meaning of any other character following a '\' is reserved for future use.

2.4.2.1 NameComponent Separators

If a name component contains a '/' slash character, the stringified representation uses the '\' character as an escape. For example, the stringified name

a/x\/y\/z/b

represents the name consisting of the name components "a," "x/y/z," and "b."

2.4.2.2 Id and kind Fields

The backslash escape mechanism is also used for '.', so **id** and **kind** fields can contain a literal '.'. To illustrate, the stringified name

a\.b.c\.d/e.f

represents the **CosNaming::Name**:

Index	id	kind
0	**a.b**	**c.d**
1	**e**	**f**

2.4.2.3 The Escape Character

The escape character '\' must be escaped if it appears in a name component. For example, the stringified name:

a/b\\/c

represents the name consisting of the components "**a**," "**b**," and "**c**."

2.5 URL schemes

This section describes the Uniform Resource Locator (URL) schemes available to represent a CORBA object and a CORBA object bound in a **NamingContext**.

2.5.1 IOR

The string form of an IOR (**IOR:<hex_octets>**) is a valid URL. The scheme name is **IOR** and the text after the ':' is defined in the CORBA 2.3 specification, Section 13.6.6. The IOR URL is robust and insulates the client from the encapsulated transport information and object key used to reference the object. This URL format is independent of Naming Service.

2.5.2 corbaloc

It is difficult for humans to exchange IORs through non-electronic means because of their length and the text encoding of binary information. The **corbaloc** URL scheme provides URLs that are familiar to people and similar to **ftp** or **http** URLs.

The **corbaloc** URL is described in the CORBA 2.3 Specification, Section 13.6.6. This URL format is independent of the Naming Service.

2.5.3 corbaname

A **corbaname** URL is similar to a **corbaloc** URL. However a corbaname URL also contains a stringified name that identifies a binding in a naming context.

2.5.3.1 corbaname Examples

corbaname::555xyz.com/dev/NContext1#a/b/c

This example denotes a naming context that can be contacted in the same manner as a **corbaloc** URL at 555xyz.com with a key of “dev/NContext1”. The “#” character denotes the start of the stringified name ,“**a/b/c**” . This name is resolved against the context to yield the final object.

corbaname::555xyz.com#a/b/c

When an object key is not specified, as in the above example, the default key of “NameService” is used to contact the naming context.

corbaname:rir:#a/b/c

This URL will resolve the stringified name “a/b/c” against the naming context returned by **resolve_initial_references(“NameService”)**.

corbaname:rir:

corbaname:rir:/NameService

The above URLs are equivalent to **corbaloc:rir:** and reference the naming context returned by **resolve_initial_references(“NameService”)**.

corbaname:atm:00033...#a/b/c

corbaname::55xyz.com,atm:00033.../dev/NCtext#a/b/c

These last URLs illustrate support of multiple protocols as allowed by **corbaloc** URLs. **atm:** is an example only and is not a defined URL protocol at this time.

Note – Unlike stringified names, **corbaname**s cannot be compared directly for equality as the address specification can differ for **corbaname** URLs with the same meaning.

2.5.3.2 *corbaname Syntax*

The full **corbaname** BNF is:

<corbaname> = “corbaname:”<corbaloc_obj>[“#”<string_name>]

<corbaloc_obj> = <obj_addr_list> [“/”<key_string>]

<obj_addr_list> = as defined in a corbaloc URL

<key_string> = as defined in a corbaloc URL

<string_name>= stringified Name | empty_string

Where:

corbaloc_obj: portion of a corbaname URL that identifies the naming context. The syntax is identical to its use in a corbaloc URL.

obj_addr_list: as defined in a corbaloc URL.

key_string: as defined in a corbaloc URL.

string_name: a stringified Name with URL escapes as defined below.

2.5.3.3 *corbaname Character Escapes*

corbaname URLs use the escape mechanism described in the Internet Engineering Task Force (IETF) RFC 2396. These escape rules insure that URLs can be transferred via a variety of transports without undergoing changes. The character escape rules for the stringified name portion of a **corbaname** are:

US-ASCII alphanumeric characters are not escaped. Characters outside this range are escaped, except for the following:

“;” | “/” | “:” | “?” | “@” | “&” | “=” | “+” | “$” |
“,” | “-” | “_” | ”.” | “!” | “~” | “*” | “’” | “(“ | “)”

2.5.3.4 *corbaname Escape Mechanism*

The percent ‘%’ character is used as an escape. If a character that requires escaping is present in a name component it is encoded as two hexadecimal digits following a “%” character to represent the octet. (The first hexadecimal character represent the high-order nibble of the octet, the second hexadecimal character represents the low-order nibble.) If a ‘%’ is not followed by two hex digits, the stringified name is syntactically invalid.

2.5.3.5 *Examples*

Stringified Name	**After URL Escapes**	**Comment**
a.b/c.d	a.b/c.d	URL form identical
<a>.b/c.d	%3ca%3e.b/c.d	Escaped “<“ and “>”
a.b/ c.d	a.b/%20%20c.d	Escaped two “ “ spaces
a%b/c%d	a%25b/c%25d	Escaped two “%” percents
a\\b/c.d	a%5c%5cb/c.d	Escaped “\” character, which is already escaped in the stringified name

2.5.3.6 *corbaname Resolution*

corbaname resolution can be implemented as a simple extension to **corbaloc** URL processing. Given a **corbaname**:

corbaname:<corbaloc_obj>[“#” <string_name>]

The **corbaname** is resolved by:

1. First constructing a **corbaloc** URL of the form: **corbaloc:<corbaloc_obj>**.

   If the **<corbaloc_obj>** does not contain a key string, a default key of “NameService” is used.

2. This is converted to a naming context object reference with **CORBA::ORB::string_to_object**.
3. The **<string_name>** is converted to a **CosNaming::Name**.
4. The resulting name is passed to a **resolve** operation on the naming context.
5. The object reference returned by the **resolve** is the result.

Implementations are not required to use the method described and may make optimizations appropriate to their environment.

2.5.4 Converting between CosNames, Stringified Names, and URLs

The **NamingContextExt** interface, derived from **NamingContext**, provides the operations required to use URLs and stringified names.

```
module CosNaming {
    // ...
    interface NamingContextExt: NamingContext {
        typedef string StringName;
        typedef string Address;
        typedef string URLString;

        StringName          to_string(in Name n) raises(InvalidName);
        Name                to_name(in StringName sn)
                        raises(InvalidName);

        exception InvalidAddress {};

        URLString           to_url(in Address addr, in StringName sn)
                        raises(InvalidAddress, InvalidName);

        Object      resolve_str(in StringName sn)
                        raises(
                            NotFound, CannotProceed,
                            InvalidName
        );
    };
};
```

2.5.4.1 to_string

This operation accepts a **Name** and returns a stringified name. If the **Name** is invalid, an InvalidName exception is raised.

2.5.4.2 to_name

This operation accepts a stringified name and returns a **Name**. If the stringified name is syntactically malformed or violates an implementation limit, an InvalidName exception is raised.

2.5.4.3 resolve_str

This is a convenience operation that performs a resolve in the same manner as **NamingContext::resolve**. It accepts a stringified name as an argument instead of a **Name**.

2.5.4.4 to_url

This operation takes a corbaloc URL **<address>** and **<key_string>** component such as

- **:myhost.555xyz.com**
- **:myhost.555xyz.com/a/b/c**
- **atm:00002112...,:myhost.xyz.com/a/b/c**

for the first parameter, and a stringified name for the second. It then performs any escapes necessary on the parameters and returns a fully formed URL string. An exception is raised if either the corbaloc address and key parameter or name parameter are malformed.

It is legal for the stringified_name to be empty. If the address is empty, an InvalidAddress exception is raised.

2.5.4.5 URL to Object Reference

Conversions from URLs to objects are handled by **CORBA::ORB::string_to_object** as described in the CORBA 2.3 Specification, Section 13.6.6.

2.6 Initial Reference to a NamingContextExt

An initial reference to an instance of this interface can be obtained by calling **resolve_initial_references** with an **ObjectID** of **NameService**.

Lightweight Naming Service 3

Note – This chapter is based on the Lightweight Services specification (ptc/04-07-03).

Contents

This chapter contains the following topics.

3.1 Platform Independent Model

3.1.1 Overview

This section defines the Platform Independent Model (PIM) for the Lightweight Naming Service. The Lightweight Naming Service is intended to be a subset of the Naming Service Specification. The packages, interfaces, and classes appearing in this chapter are intended to model this subset and should map to the IDL for their counterparts in the Naming Service Specification (Version 1.2, September 2002, formal/02-09-02). The descriptions of the interfaces, operations and their semantics are also intended to be identical to those defined by the Naming Service Specification (Version 1.2, September 2002, formal/02-09-02) over this same subset.

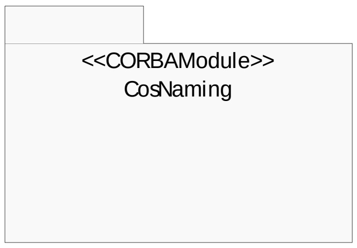


Figure 3-1 - Lightweight Naming Service Packages

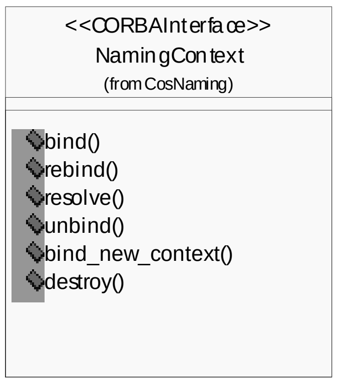


Figure 3-2 - Lightweight Naming Service Interfaces and Classes

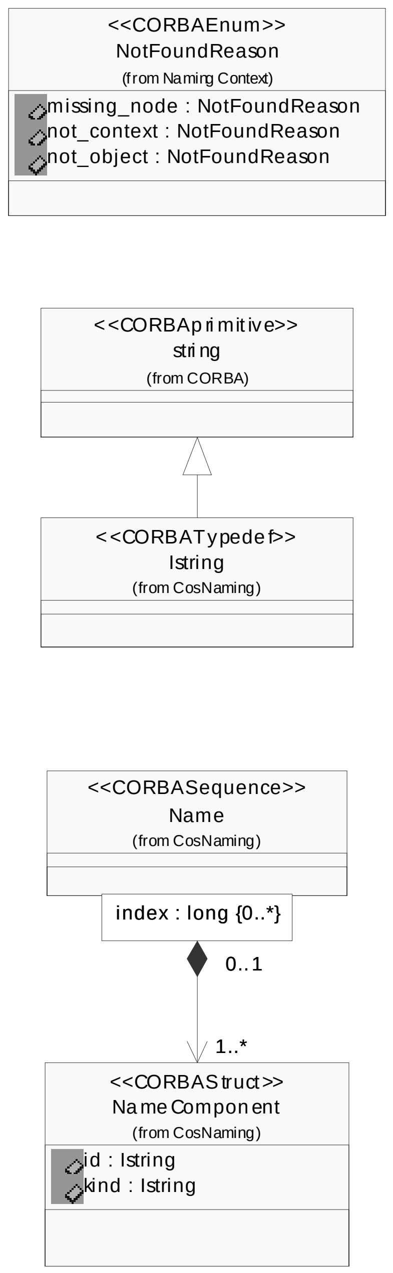


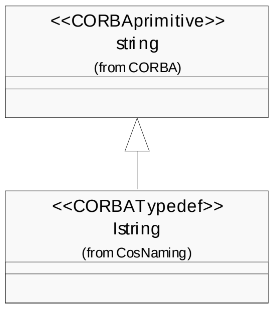
Figure 3-3 - Lightweight Naming Service Data Types

3.1.2 The CosLightweightNaming Package

The **CosLightweightNaming** package is a collection of interfaces, datatypes, and exceptions that together define the Lightweight Naming Service. Unlike the full **CosNamingService**, this package supports only the **NamingContext** interface.

3.1.2.1 Istring

Description



Istring is a "placeholder for a future IDL internationalized string data type" in the original **CosNaming** specification. It is maintained solely for compatibility reasons.

Attributes

No additional attributes

Operations

No additional operations

Associations

No associations

Constraints

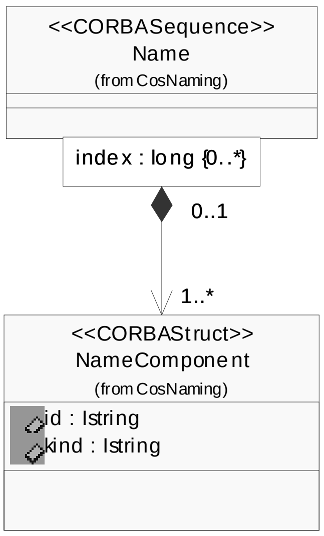
No additional constraints

Semantics

No additional semantics

3.1.2.2 Name

Description



A name is a sequence of **NameComponents**.

Attributes

No attributes

Operations

No operations

Associations

`component: NameComponent[1..*]`

A name consists of an ordered list of NameComponents.

Constraints

No constraints

Semantics

A name is a sequence of **NameComponents**. The empty sequence is not a legal name. An implementation may limit the length of the sequence to some maximum. When comparing Names for equality, each **NameComponent** in the first name must match the corresponding **NameComponent** in the second Name for the names to be considered identical.

3.1.2.3 NameComponent

Description

The **NameComponent** represents one segment of the name, consisting of two parts represented as attributes.

Attributes

`id: Istring  [1]`

An arbitrary length string holding the main component of the name.
(Comment:This is usually the name iteslf.)

`kind: Istring  [1]`

An arbitrary length string holding the additonal component of the name.
(Comment: This is usually some characterization of the name.)

Operations

No operations

Associations

No associations

Constraints

No constraints

Semantics

A name component consists of two attributes: the identifier attribute, id, and the kind attribute, kind.

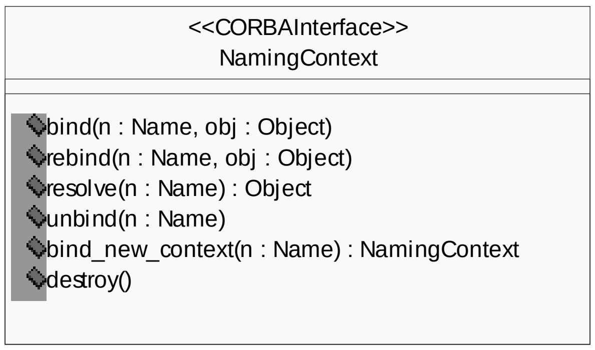
Both of these attributes are arbitrary-length strings of ISO Latin-1 characters, excluding the ASCII NUL character.

When comparing two NameComponents for equality both the id and the kind field must match in order for two NameComponents to be considered identical. This applies for zero-length (empty) fields as well. Name comparisons are case sensitive.

An implementation may place limitations on the characters that may be contained in a name component, as well as the length of a name component. For example, an implementation may disallow certain characters, may not accept the empty string as a legal name component, or may limit name components to some maximum length.

3.1.2.4 NamingContext

Description


<<CORBAInterface>>
NamingContext
bind(n : Name, obj : Object)
rebind(n : Name, obj : Object)
resolve(n : Name) : Object
unbind(n : Name)
bind_new_context(n : Name) : NamingContext
destroy()

A NamingContext is a container hosting a set of name bindings.

Attributes

No attributes.

Operations

`bind(in n: Name, in obj: Object)`

Creates an object binding in the naming context. If a binding with the specified name already exists, bind will raise an AlreadyBound exception. If an implementation places limits on the number of bindings within a context, bind will raise the IMP_LIMIT system exception if the new binding cannot be created. The operation may also raise NotFound, CannotProceed, or InvalidName.

`rebind(in n: Name, in obj: Object)`

Creates an object binding in the naming context even if the name is already bound in the context. If already bound, the previous binding must be of type object; otherwise, a NotFound exception with a why reason of **not_object** is raised. If rebind raises a NotFound exception because an already existing binding is of the wrong type, the **rest_of_name** member of the exception has a sequence length of 1. The operation may also raise CannotProceed or InvalidName.

`resolve (in n: Name): Object)`

The resolve operation retrieves an object bound to a name in a given context. The given name must exactly match the bound name. The naming service does not return the type of the object. Clients are responsible for "narrowing" the object to the appropriate type. That is, clients typically cast the returned object from Object to a more specialized interface.

Names can have multiple components; therefore, name resolution can traverse multiple contexts. These contexts can be federated between different Naming Service instances. The operation may raise NotFound, CannotProceed, or InvalidName.

`unbind(in n: Name)`

The unbind operation removes a name binding from a context. The operation may raise NotFound, CannotProceed, or InvalidName.

`bind_new_context (in n: Name): NamingContext`

This operation creates a new context and creates an context binding for it using the name supplied as an argument.

If an implementation places limits on the number of naming contexts, **bind_new_context** can raise the IMP_LIMIT system exception if the context cannot be created. **bind_new_context** can also raise IMP_LIMIT if the bind would cause an implementation limit on the number of bindings in a context to be exceeded.

The operation may also raise NotFound, CannotProceed, or InvalidName.

`destroy()`

This operation destroys its naming context. If there are bindings denoting the destroyed context, these bindings are not removed. If the naming context contains bindings, the operation raises NotEmpty.

Associations

No association.

Constraints

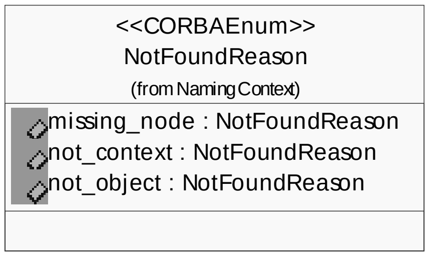
No constraints.

Semantics

A name-to-object association is called a name binding. A name binding is always defined relative to a naming context. A naming context is an object that contains a set of name bindings in which each name is unique. Different names can be bound to an object in the same or different contexts at the same time. There is no requirement, however, that all objects must be named. To resolve a name is to determine the object associated with the name in a given context. To bind a name is to create a name binding in a given context. A name is always resolved relative to a context - there are no absolute names. Because a context is like any other object, it can also be bound to a name in a naming context. Binding contexts in other contexts creates a naming graph - a directed graph with nodes and labeled edges where the nodes are contexts. A naming graph allows more complex names to reference an object. Given a context in a naming graph, a sequence of names can reference an object. This sequence of names (called a compound name) defines a path in the naming graph to navigate the resolution process.

3.1.2.5 NamingContext::NotFoundReason

Description


<<CORBAEnum>>
NotFoundReason
(from NamingContext)
missing_node : NotFoundReason
not_context : NotFoundReason
not_object : NotFoundReason

The enumeration **NotFoundReason** specifies the reason that a NotFound exception was raised with respect to resolution of a given name (which may be a component of a larger name).

Attributes

`missing_node`

The first component of the given name is not bound within its parent context.

`not_context`

The first name component of the given name denotes a binding with a type of nobject when the type ncontext was required.

`not_object`

The first name component of the given name denotes a binding with a type of ncontext when the type nobject was required.

Operations

No operations

Associations

No associations

Constraints

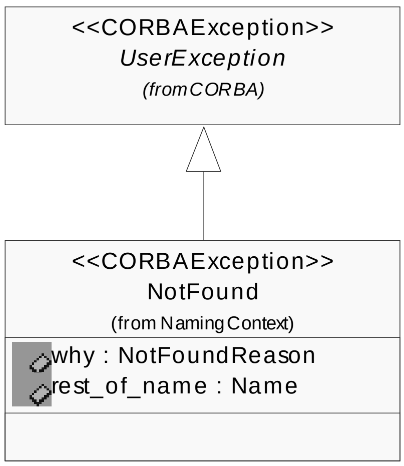
No constraints

Semantics

This is an Enumeration type.

3.1.2.6 NamingContext::NotFound

Description



The NotFound user exception.

Attributes

`why: NotFoundReason [1]`

The why attribute explains the reason for the exception.

`rest_of_name: Name [1]`

The **rest_of_name** attribute contains the remainder of the non-working name:

Operations

No operations

Associations

No associations

Constraints

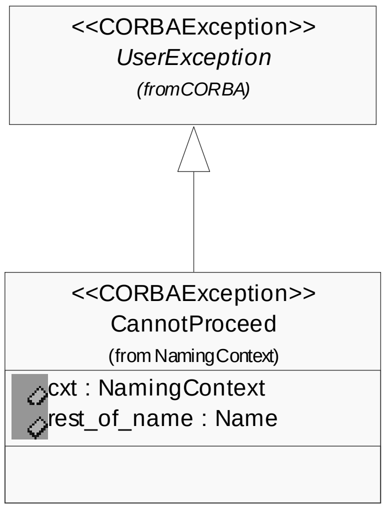
No constraints

Semantics

This exception is raised by operations when a component of a name does not identify a binding, or the type of the binding is incorrect for the operation being performed.

3.1.2.7 NamingContext::CannotProceed

Description



The CannotProceed user exception.

Attributes

`cxt: NamingContext [1]`

The cxt attribute contains the context that the operation may be able to retry from.

`rest_of_name: Name [1]`

The **rest_of_name** attribute contains the remainder of the non-working name:

Operations

No operations

Associations

No associations.

Constraints

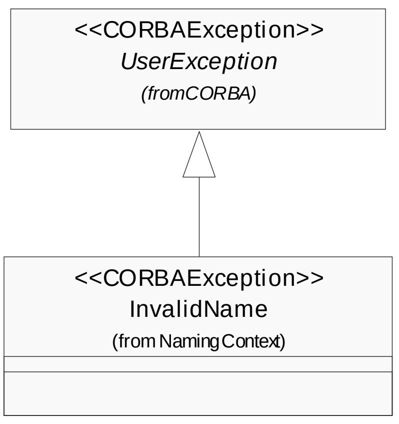
No constraints.

Semantics

This exception is raised when an implementation has given up for some reason. The client, however, may be able to continue the operation at the returned naming context.

3.1.2.8 NamingContext::InvalidName

Description



The InvalidName user exception.

Attributes

No attributes.

Operations

No operation.

Constraints

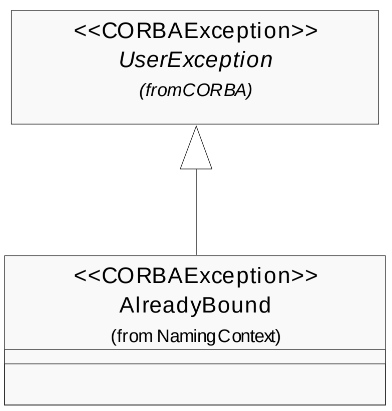
No constraints.

Semantics

This exception is raised if a Name is invalid. A name of length zero is invalid (containing no name components). Implementations may place further limitations on what constitutes a legal name and raise this exception to indicate a violation.

3.1.2.9 NamingContext::AlreadyBound

Description



The AlreadyBound user exception.

Attributes

No attributes.

Operations

No operation.

Constraints

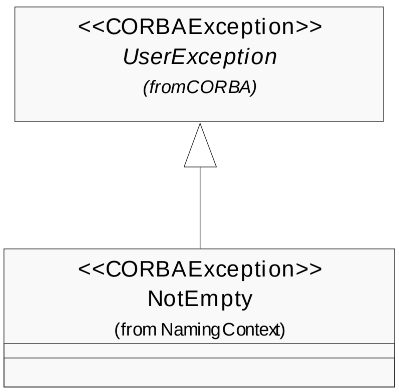
No constraints.

Semantics

Indicates an object is already bound to the specified name. Only one object can be bound to a particular Name in a context. The lightweight naming service user must use the “rebind” interface to explicitly bind a newobject reference to an existing name.

3.1.2.10 NamingContext::NotEmpty

Description



The NotEmpty user exception.

Attributes

No attributes.

Operations

No operation.

Constraints

No constraints.

Semantics

This exception is raised by destroy if the **NamingContext** contains bindings. A **NamingContext** must be empty to be destroyed.

3.2 Platform Specific Model: CORBA Service

3.2.1 Overview

The following sections specify a platform specific mapping of the Lightweight Naming Service onto the CORBA platform. The resulting CORBA service is specified in CORBA IDL and represents a fully compatible subset of the **CosNamingService**.

3.2.2 CosNaming Module

```
#ifndef _COSNAMING_IDL_
#define _COSNAMING_IDL_

#ifdef _PRE_3_0_COMPILER_
# pragma prefix "omg.org"
#endif

module CosNaming
{
# ifndef _PRE_3_0_COMPILER_
   typeprefix "omg.org";
# endif // _PRE_3_0_COMPILER_
```

3.2.2.1 Istring

```
   typedef string Istring;
```

3.2.2.2 NameComponent

```
   struct NameComponent
   {
      Istring id;
      Istring kind;
   };
   typedef sequence<NameComponent> Name;
```

3.2.2.3 NamingContext

```
   interface NamingContext
   {

      enum NotFoundReason { missing_node, not_context, not_object };

      exception NotFound
         {
         NotFoundReason why;
         Name rest_of_name;
      };

      exception CannotProceed
      {
         NamingContext cxt;
         Name rest_of_name;
      };

      exception InvalidName  {};
      exception AlreadyBound {};
      exception NotEmpty     {};
```

```
        void bind(in Name n, in Object obj)
            raises(NotFound, CannotProceed, InvalidName, AlreadyBound);
        void rebind(in Name n, in Object obj)
            raises(NotFound, CannotProceed, InvalidName);
        Object resolve (in Name n)
            raises(NotFound, CannotProceed, InvalidName);
        void unbind(in Name n)
            raises(NotFound, CannotProceed, InvalidName);
        NamingContext bind_new_context(in Name n)
            raises(NotFound, AlreadyBound, CannotProceed, InvalidName);
        void destroy()
            raises(NotEmpty);
    };

};
#endif // _COSNAMING_IDL_
```

OMG IDL A

```
// File: CosNaming.idl
#ifndef _COSNAMING_IDL_
#define _COSNAMING_IDL_

#pragma prefix "omg.org"

module CosNaming {
    typedef string Istring;

    struct NameComponent {
        Istring id;
        Istring kind;
    };
    typedef sequence<NameComponent> Name;

    enum BindingType { nobject, ncontext };

    struct Binding {
        Name              binding_name;
        BindingType       binding_type;
    };

    // Note: In struct Binding, binding_name is incorrectly defined
    // as a Name instead of a NameComponent. This definition is
    // unchanged for compatibility reasons.
    typedef sequence <Binding> BindingList;

    interface BindingIterator;

    interface NamingContext {
        enum NotFoundReason { missing_node, not_context, not_object };

        exception NotFound {
            NotFoundReason       why;
            Name                 rest_of_name;
        };
        exception CannotProceed {
            NamingContext           cxt;
            Name                    rest_of_name;
        };

        exception InvalidName{};

        exception AlreadyBound {};

        exception NotEmpty{};

        void    bind(in Name n, in Object obj)
                    raises(
                        NotFound, CannotProceed, InvalidName, AlreadyBound
                    );

        void    rebind(in Name n, in Object obj)
                    raises(NotFound, CannotProceed, InvalidName);

        void    bind_context(in Name n, in NamingContext nc)
                    raises(NotFound, CannotProceed, InvalidName, AlreadyBound);

        void    rebind_context(in Name n, in NamingContext nc)
                    raises(NotFound, CannotProceed, InvalidName);

        Object  resolve (in Name n)
                    raises(NotFound, CannotProceed, InvalidName);

        void    unbind(in Name n)
                    raises(NotFound, CannotProceed, InvalidName);

        NamingContext   new_context();
        NamingContext   bind_new_context(in Name n)
                            raises(
                                NotFound, AlreadyBound,
                                CannotProceed, InvalidName
                            );

        void    destroy() raises(NotEmpty);

        void    list(
                 in unsigned long           how_many,
                 out BindingList            bl,
                 out BindingIterator        bi
            );

    };

    interface BindingIterator {
        boolean next_one(out Binding b);
        boolean next_n(in unsigned long how_many, out BindingList bl);
        void    destroy();
    };
```

```
    interface NamingContextExt: NamingContext {
        typedef string StringName;
        typedef string Address;
        typedef string URLString;

        StringName       to_string(in Name n) raises(InvalidName);
        Name             to_name(in StringName sn)
                                raises(InvalidName);

        exception InvalidAddress {};

        URLString   to_url(in Address addr, in StringName sn)
                                raises(InvalidAddress, InvalidName);

        Object     resolve_str(in StringName sn)
                                raises(
                                       NotFound, CannotProceed,
                                       InvalidName,
                                );

        };
};

#endif // _COSNAMING_IDL_
```

Conformance Requirements B

B.1 Optional Interfaces

There are no optional interfaces in this specification. A compliant implementation must implement all of the functionality and interfaces described.

B.2 Documentation Requirements

A compliant implementation must document all of the following:

- any limitations to the character values or character sequences that may be used in a name component
- any limitations to the length (including minimum or maximum) of a name component
- any limitations to number of name components in a name
- any limitations to the maximum number of bindings in a context
- any limitations to the total number of bindings (implementation-wide)
- any limitations to the maximum number of contexts
- the means provided to deal with orphaned contexts and bindings
- Any policy for dealing with potentially orphaned naming contexts. Orphaned contexts are contexts that are not bound in any other context within a naming server.
- Any policy for destroying binding iterators that are considered to be no longer in use.
- Under what circumstances, if any, a CannotProceed exception is raised.

Index

Index