
Externalization Service Specification

Version 1.0
New Edition: April 2000

Copyright 1994 International Business Machines Corporation
Copyright 1994 SunSoft, Inc.

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

PATENT

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

NOTICE

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

WHILE THE INFORMATION IN THIS PUBLICATION IS BELIEVED TO BE ACCURATE, THE OBJECT MANAGEMENT GROUP AND THE COMPANIES LISTED ABOVE MAKE NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS MATERIAL INCLUDING, BUT NOT LIMITED TO ANY WARRANTY OF TITLE OR OWNERSHIP, IMPLIED WARRANTY OF MERCHANTABILITY OR WARRANTY OF FITNESS FOR PARTICULAR PURPOSE OR USE. In no event shall The Object Management Group or any of the companies listed above be liable for errors contained herein or for indirect, incidental, special, consequential, reliance or cover damages, including loss of profits, revenue, data or use, incurred by any user or any third party. The copyright holders listed above acknowledge that the Object Management Group (acting itself or through its designees) is and shall at all times be the sole entity that may authorize developers, suppliers and sellers of computer software to use certification marks, trademarks or other special designations to indicate compliance with these materials. This document contains information which is protected by copyright. All Rights Reserved. No part of this work covered by copyright herein may be reproduced or used in any form or by any means--graphic, electronic, or mechanical, including photocopying, recording, taping, or information storage and retrieval systems--without permission of the copyright owner.

RESTRICTED RIGHTS LEGEND. Use, duplication, or disclosure by government is subject to restrictions as set forth in subdivision (c) (1) (ii) of the Right in Technical Data and Computer Software Clause at DFARS 252.227.7013. OMG[®] and Object Management are registered trademarks of the Object Management Group, Inc. Object Request Broker, OMG IDL, ORB, CORBA, CORBA facilities, CORBA services, COSS, and IIOP are trademarks of the Object Management Group, Inc. X/Open is a trademark of X/Open Company Ltd.

ISSUE REPORTING

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

Contents

Preface	iii
About the Object Management Group	iii
What is CORBA?	iii
Associated OMG Documents	iv
Acknowledgments	iv
1. Service Description	1-1
1.1 Overview	1-1
1.2 Service Structure	1-2
1.2.1 Client's Model of Object Externalization	1-2
1.2.2 Stream's Model of Object Externalization ...	1-3
1.2.3 Object's Model of Externalization	1-4
1.2.4 Object's Model of Internalization	1-6
1.3 Object and Interface Hierarchies	1-8
1.4 Interface Summary	1-11
1.4.1 Externalization Service Architecture: Audience/Bearer Mapping	1-12
2. Externalization Service Modules	2-1
2.1 CosExternalization Module	2-1
2.1.1 StreamFactory Interface	2-2
2.1.2 FileStreamFactory Interface	2-3
2.1.3 Stream Interface	2-3
2.2 CosStream Module	2-5
2.2.1 Standard Stream Data Format	2-8
2.2.2 The StreamIO Interface	2-9

Contents

2.2.3	The Streamable Interface	2-9
2.2.4	The StreamableFactory Interface	2-11
2.2.5	The Node Interface	2-12
2.2.6	The Role Interface	2-13
2.2.7	The Relationship Interface	2-13
2.2.8	The PropagationCriteriaFactory Interface.	2-14
2.3	Specific Externalization Relationships	2-15
2.4	The CosExternalizationContainment Module.	2-15
2.5	The CosExternalizationReference Module	2-16
2.6	Standard Stream Data Format.	2-18
2.6.1	OMG Externalized Object Data	2-18
2.6.2	Externalized Repeated Reference Data.	2-19
2.6.3	Externalized NIL Data	2-20
	Appendix A - References.	A-1

Preface

About This Document

Under the terms of the collaboration between OMG and X/Open Co Ltd, this document is a candidate for endorsement by X/Open, initially as a Preliminary Specification and later as a full CAE Specification. The collaboration between OMG and X/Open Co Ltd ensures joint review and cohesive support for emerging object-based specifications.

X/Open Preliminary Specifications undergo close scrutiny through a review process at X/Open before publication and are inherently stable specifications. Upgrade to full CAE Specification, after a reasonable interval, takes place following further review by X/Open. This further review considers the implementation experience of members and the full implications of conformance and branding.

Object Management Group

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

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

What is CORBA?

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

X/Open

X/Open is an independent, worldwide, open systems organization supported by most of the world's largest information system suppliers, user organizations and software companies. Its mission is to bring to users greater value from computing, through the practical implementation of open systems.

Intended Audience

The specifications described in this manual are aimed at software designers and developers who want to produce applications that comply with OMG standards for object services; the benefits of compliance are outlined in the following section, "Need for Object Services."

Need for Object Services

To understand how Object Services benefit all computer vendors and users, it is helpful to understand their context within OMG's vision of object management. The key to understanding the structure of the architecture is the Reference Model, which consists of the following components:

- **Object Request Broker**, which enables objects to transparently make and receive requests and responses in a distributed environment. It is the foundation for building applications from distributed objects and for interoperability between applications in hetero- and homogeneous environments. The architecture and specifications of the Object Request Broker are described in *CORBA: Common Object Request Broker Architecture and Specification*.
- **Object Services**, a collection of services (interfaces and objects) that support basic functions for using and implementing objects. Services are necessary to construct any distributed application and are always independent of application domains.
- **Common Facilities**, a collection of services that many applications may share, but which are not as fundamental as the Object Services. For instance, a system management or electronic mail facility could be classified as a common facility.

The Object Request Broker, then, is the core of the Reference Model. Nevertheless, an Object Request Broker alone cannot enable interoperability at the application semantic level. An ORB is like a telephone exchange: it provides the basic mechanism for making and receiving calls but does not ensure meaningful communication between subscribers. Meaningful, productive communication depends on additional interfaces, protocols, and policies that are agreed upon outside the telephone system, such as telephones, modems and directory services. This is equivalent to the role of Object Services.

What Is an Object Service Specification?

A specification of an Object Service usually consists of a set of interfaces and a description of the service's behavior. The syntax used to specify the interfaces is the OMG Interface Definition Language (OMG IDL). The semantics that specify a services's behavior are, in general, expressed in terms of the OMG Object Model. The OMG Object Model is based on objects, operations, types, and subtyping. It provides a standard, commonly understood set of terms with which to describe a service's behavior.

(For detailed information about the OMG Reference Model and the OMG Object Model, refer to the *Object Management Architecture Guide*).

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*.)

To obtain print-on-demand books in the documentation set or other OMG publications, contact the Object Management Group, Inc. at:

OMG Headquarters
250 First Avenue, Suite 201
Needham, MA 02494
USA
Tel: +1-781-444-0404
Fax: +1-781-444-0320
pubs@omg.org
<http://www.omg.org>

Service Design Principles

Build on CORBA Concepts

The design of each Object Service uses and builds on CORBA concepts:

- Separation of interface and implementation
- Object references are typed by interfaces
- Clients depend on interfaces, not implementations
- Use of multiple inheritance of interfaces
- Use of subtyping to extend, evolve and specialize functionality

Other related principles that the designs adhere to include:

- Assume good ORB and Object Services implementations. Specifically, it is assumed that CORBA-compliant ORB implementations are being built that support efficient local and remote access to “fine-grain” objects and have performance characteristics that place no major barriers to the pervasive use of distributed objects for virtually all service and application elements.
- Do not build non-type properties into interfaces

A discussion and rationale for the design of object services was included in the HP-SunSoft response to the OMG Object Services RFI (OMG TC Document 92.2.10).

Basic, Flexible Services

The services are designed to do one thing well and are only as complicated as they need to be. Individual services are by themselves relatively simple yet they can, by virtue of their structuring as objects, be combined together in interesting and powerful ways.

For example, the event and life cycle services, plus a future relationship service, may play together to support graphs of objects. Object graphs commonly occur in the real world and must be supported in many applications. A functionally-rich Folder compound object, for example, may be constructed using the life cycle, naming, events, and future relationship services as “building blocks.”

Generic Services

Services are designed to be generic in that they do not depend on the type of the client object nor, in general, on the type of data passed in requests. For example, the event channel interfaces accept event data of any type. Clients of the service can dynamically determine the actual data type and handle it appropriately.

Allow Local and Remote Implementations

In general the services are structured as CORBA objects with OMG IDL interfaces that can be accessed locally or remotely and which can have local library or remote server styles of implementations. This allows considerable flexibility as regards the location of participating objects. So, for example, if the performance requirements of a particular application dictate it, objects can be implemented to work with a Library Object Adapter that enables their execution in the same process as the client.

Quality of Service is an Implementation Characteristic

Service interfaces are designed to allow a wide range of implementation approaches depending on the quality of service required in a particular environment. For example, in the Event Service, an event channel can be implemented to provide fast but unreliable delivery of events or slower but guaranteed delivery. However, the interfaces to the event channel are the same for all implementations and all clients. Because rules are not wired into a complex type hierarchy, developers can select particular implementations as building blocks and easily combine them with other components.

Objects Often Conspire in a Service

Services are typically decomposed into several distinct interfaces that provide different views for different kinds of clients of the service. For example, the Event Service is composed of *PushConsumer*, *PullSupplier* and *EventChannel* interfaces. This simplifies the way in which a particular client uses a service.

A particular service implementation can support the constituent interfaces as a single CORBA object or as a collection of distinct objects. This allows considerable implementation flexibility. A client of a service may use a different object reference to communicate with each distinct service function. Conceptually, these “internal” objects *conspire* to provide the complete service.

As an example, in the Event Service an event channel can provide both *PushConsumer* and *EventChannel* interfaces for use by different kinds of client. A particular client sends a request not to a single “event channel” object but to an object that implements either the *PushConsumer* and *EventChannel* interface. Hidden to all the clients, these objects interact to support the service.

The service designs also use distinct objects that implement specific service interfaces as the means to distinguish and coordinate different clients without relying on the existence of an object equality test or some special way of identifying clients. Using the event service again as an example, when an event consumer is connected with an event channel, a new object is created that supports the *PullSupplier* interface. An object reference to this object is returned to the event consumer which can then request events by invoking the appropriate operation on the new “supplier” object. Because each client uses a different object reference to interact with the event channel, the event channel can keep track of and manage multiple simultaneous clients. An event channel as a collection of objects conspiring to manage multiple simultaneous consumer clients.

Use of Callback Interfaces

Services often employ callback interfaces. Callback interfaces are interfaces that a client object is required to support to enable a service to *call back* to it to invoke some operation. The callback may be, for example, to pass back data asynchronously to a client.

Callback interfaces have two major benefits:

- They clearly define how a client object participates in a service.
- They allow the use of the standard interface definition (OMG IDL) and operation invocation (object reference) mechanisms.

Assume No Global Identifier Spaces

Several services employ identifiers to label and distinguish various elements. The service designs do not assume or rely on any global identifier service or global id spaces in order to function. The scope of identifiers is always limited to some context. For example, in the naming service, the scope of names is the particular naming context object.

In the case where a service generates ids, clients can assume that an id is unique within its scope but should not make any other assumption.

Finding a Service is Orthogonal to Using It

Finding a service is at a higher level and orthogonal to using a service. These services do not dictate a particular approach. They do not, for example, mandate that all services must be found via the naming service. Because services are structured as objects there does not need to be a special way of finding objects associated with services - general purpose finding services can be used. Solutions are anticipated to be application and policy specific.

Interface Style Consistency

Use of Exceptions and Return Codes

Throughout the services, exceptions are used exclusively for handling exceptional conditions such as error returns. Normal return codes are passed back via output parameters. An example of this is the use of a DONE return code to indicate iteration completion.

Explicit Versus Implicit Operations

Operations are always explicit rather than implied (e.g., by a flag passed as a parameter value to some “umbrella” operation). In other words, there is always a distinct operation corresponding to each distinct function of a service.

Use of Interface Inheritance

Interface inheritance (subtyping) is used whenever one can imagine that client code should depend on less functionality than the full interface. Services are often partitioned into several unrelated interfaces when it is possible to partition the clients into different roles. For example, an administrative interface is often unrelated and distinct in the type system from the interface used by “normal” clients.

Acknowledgments

The following companies submitted parts of the *Externalization Service* specification:

- International Business Machines Corporation
- SunSoft, Inc.

Service Description

1

Contents

This chapter contains the following topics.

Topic	Page
“Overview”	1-1
“Service Structure”	1-2
“Object and Interface Hierarchies”	1-8
“Interface Summary”	1-11

Note – Dec. 1998: OMG made some editorial changes. These changes involved merging the old CosCompoundExternalization section and IDL into the CosStream section and IDL. Those were the only changes, along with a few text references to CosCompoundExternalization that became CosStream, which have been marked with changebars.

1.1 Overview

The Externalization Service specification defines protocols and conventions for externalizing and internalizing objects. To externalize an object is to record the object’s state in a stream of data. Objects which support the appropriate interfaces and whose implementations adhere to the proper conventions can be externalized to a stream (in memory, on a disk file, across the network, etc.) and subsequently be internalized into a new object in the same or a different process. The externalized form of the object can exist for arbitrary amounts of time, be transported by means outside of the ORB, and can be internalized in a different, disconnected ORB.

Many different externalized data formats and storage mediums can be supported by service implementations. But, for portability, clients can request that externalized data be stored in a file using a standardized format that is defined as part of this Externalization Service specification.

Externalizing and internalizing an object is similar to copying the object. The copy operation creates a new object that is initialized from an existing object. The new object is then available to provide service. Furthermore, with the copy operation, there is an assumption that it is possible to communicate via the ORB between the “here” and “there”. Externalization, on the other hand, does not create an object that is initialized from an existing object. Externalization “stops along the way”. New objects are not created until the stream is internalized. Furthermore, there is no assumption that is possible to communicate via the ORB between “here” and “there.”

The Externalization Service is related to the Relationship Service. It also parallels the Life Cycle Service in defining externalization protocols for simple objects, for arbitrarily related objects, and for graphs of related objects that support compound operations. (For more information, refer to the Service Dependencies section in Chapter 2.)

The Externalization Service defines protocols in these areas:

- Client’s view of externalization, composed of the interfaces used by a client to externalize and internalize objects. The client’s view of externalization is defined by the **Stream** interface.
- Object’s view of externalization, composed of the interfaces used by an externalizable object to record and retrieve their object state to and from the stream’s external form. The object’s view is defined by the **StreamIO** interface.
- Stream’s view of externalization, composed of the interfaces used by the stream to direct an externalizable object or graph of objects to record or retrieve their state from the stream’s external form. The stream’s view of externalization is given by the **Streamable**, **Node**, **Role** and **Relationship** interfaces.

1.2 Service Structure

This section explains the model of externalization for client and stream. It also describes the model of externalization and internalization for objects.

1.2.1 Client’s Model of Object Externalization

A client has a simple view of the externalization service. A client that wishes to externalize an object first must have an object reference for a **Stream** object. A **Stream** object owns and provides access to the externalized form of one or more objects. Streams may be provided that hold externalized data on various mediums such as in memory or on disk. All Externalization Service implementors provide a **Stream** object that saves the externalized data in a file. A client may create a **Stream** object using the **create()** operation on a **StreamFactory** object, or may specify that a file be used to store the externalized data using the **create()** operation of a **FileStreamFactory** object.

The client can create a **Stream** object that supports a standardized externalization data format. Externalization data that follows this format will be internalizable on all CORBA-compliant ORBs that can locate compatible object implementations. By including support for a specific external representation format in the Externalization Service, portability of object state is provided across different CORBA-compliant implementations and hardware architectures.

Once a client has a **Stream** object, the client may externalize an object by issuing an **externalize()** request on the **Stream** object, providing the object reference to the object that should be externalized. In general, the client is unaware of whether externalizing an object causes any other related objects to be externalized. An externalizable object may represent a simple object, a set of objects, or a graph of related objects. The client uses the same interface in all cases.

If a client wishes to externalize multiple objects (or related sets of objects) to the same stream, the client issues a **begin_context()** request before the first externalize request and then issues an **end_context()** following the last externalize request for that same stream.

The externalized form of the object can exist in the stream object for arbitrary amounts of time, be transported by means outside of the ORB, and can be internalized in a different, disconnected ORB.

A client that wishes to internalize an object issues an **internalize()** request on the appropriate **Stream** object, providing a factory finder. The **Stream** object interacts with the specified factory finder, or uses other implementation dependent mechanisms, to create an implementation of the object that matches the externalized data. The client is returned an object reference to the newly internalized object.

1.2.2 Stream's Model of Object Externalization

A **Stream** object provides the **Stream** interface for use by clients. The **Stream** object is also responsible for providing an object that supports a **StreamIO** interface for actually reading and writing data to the externalized data form. The stream object may support the **StreamIO** interfaces itself, or may create another object that supports the **StreamIO** interfaces. This is considered an implementation detail.

Note – When the behavior described in this section may be implemented in either the **Stream** or **StreamIO** objects (or other internal objects they may use), the term “stream service” is used.

When a stream object receives an externalize request from a client, it also gets an object reference to the object to be externalized. The stream cooperates with the externalizable object to accomplish externalization and internalization, using the object's **Streamable** interfaces.

The stream service uses the readonly **Key** attribute of the externalizable object to decide what information to put into the external data in order to be able to find the correct factory and implementation with which to subsequently internalize an equivalent object. The stream service then issues an **externalize_to_stream()** request

to the externalizable object, providing an object reference to a **StreamIO** object that is to be used by the externalizable object to record its state in the stream service's external data.

When a **Stream** object receives an internalize request from a client, it also gets a factory finder. The stream service holds the external form of the object, or set of objects, to be internalized. The stream service reads the key from its externalized data. It may then pass the key to the factory finder to locate a factory that can create an object with an implementation that matches the recorded object state. The stream service implementation may use other implementation specific ways of creating an appropriate object. The stream service then issues an **internalize_from_stream()** request to the newly created object, providing an object reference to a **StreamIO** object that is used by the externalizable object to initialize its state according to the stream service's externalized data.

When a **Stream** object receives a **begin_context()** request, the stream service sets up a context during which the stream service ensures that externalizing multiple objects that may have overlapping object references and/or object relationships produces single instances of those objects on internalization. An **end_context()** request causes the stream service to remove the previous internal context, and externalize subsequent objects without regard to whether they have already been externalized in this **Stream**'s data.

8.2.3 Object's Model of Externalization

Every object that wishes to be externalizable must support the **Streamable** interface, and follow conventions on use of the **StreamIO** interfaces to record and retrieve their object state from a **Stream**'s data.

When a **Streamable** object receives an **externalize_to_stream** request from the stream service, it must write all of its state necessary for internalization to the **StreamIO** object provided by the stream service. **StreamIO** provides **write_<type>()** operations for writing each of the CORBA basic data types, plus string types. If an object has object references that are part of its state, the **StreamIO** **write_object()** operation may be used to cause the object specified by an object reference to also be externalized to the stream's data.

Externalization Control Flow (streamable object is not a node)

Client calls **Stream::externalize(Streamable object)**



Stream writes a key for this object to the external representation.

Stream calls the **Streamable::write_to_stream(StreamIO this_sio)** so that the object can write out whatever internal state it needs to save.

If **Streamable** object is a node in a graph of related objects, flow is given in Figure 1-2

Streamable object writes out its non-object data using the primitive **StreamIO::write_...(data)** functions

Streamable object writes out other objects using the **StreamIO::write_object(Streamable object)** function

Figure 1-1 Externalization control flow when streamable object is not in a graph of related objects

A streamable object may be a node in a graph of related objects, that is, it may use the Relationship Service to connect to other objects and support the **CosStream::Node** interface. Such a streamable object simply delegates the **Streamable::externalize_to_stream()** request back to the stream service, using the **StreamIO::write_graph()** operation.

The stream service then coordinates the externalization of the graph and calls the object back using the object's **CosStream::Node** interface.

Externalization Control Flow (streamable is a node)

Streamable object, recognizing that it is a node in a graph of related objects, delegates the externalization of the graph to the stream service using **StreamIO::write_graph(this_node)** operation.

StreamIO::write_graph, coordinates the externalization of the graph using **Node::externalize_node(this_sio)** operation.

Node writes out its non-object data using the primitive **StreamIO::write...(data)** functions

Node writes out other objects using the **StreamIO::write_object(Streamable object)** function

Node writes out its role objects using the **Role::externalize_role(this_sio)** operation.

StreamIO::write_graph uses propagation value to determine next nodes and writes a key for next node

StreamIO object externalizes the involved relationships using **Relationship::externalize()**. **StreamIO** writes traversal scoped ids for the externalized roles and relationships to the **Stream**'s data.

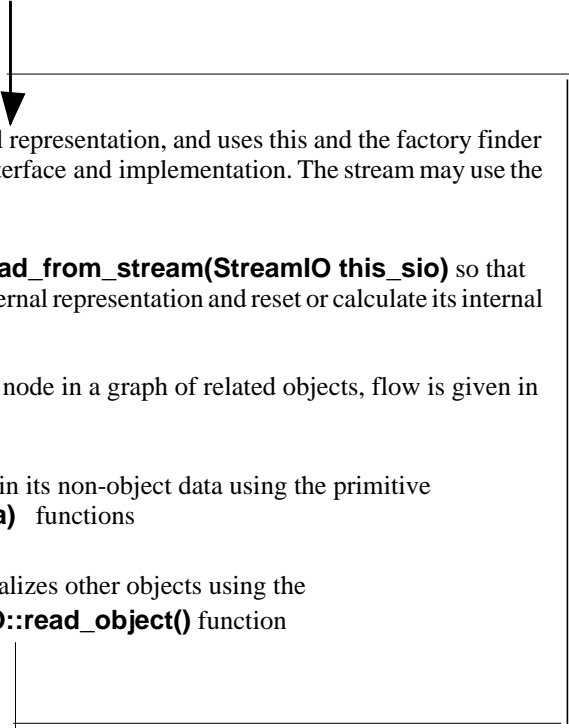
Figure 1-2 Externalization control flow when streamable object is a node in a graph of related objects

1.2.4 Object's Model of Internalization

When a streamable object receives an **internalize_from_stream()** request from a stream, it must read data from the **StreamIO** object provided by the stream service, and initialize its state to match the externalized state. The externalizable object requests data from the stream service using the **StreamIO read_<type>()** operations for basic data, and string types. If the object being internalized includes a reference to another object as part of its state, the **StreamIO read_object()** operation may be used to have that object also internalized from the stream's data.

Internalization Control Flow (streamable object is not a node)

Client calls **Streamable = Stream::internalize(FactoryFinder f)**



Stream reads key from the external representation, and uses this and the factory finder to *create an object* of the correct interface and implementation. The stream may use the **StreamableFactory** interface.

Stream calls the **Streamable::read_from_stream(StreamIO this_sio)** so that the object can read the data in its external representation and reset or calculate its internal state

If **Streamable** object is a node in a graph of related objects, flow is given in Figure 1-4

Streamable object reads in its non-object data using the primitive **StreamIO::read...(data)** functions

Streamable object internalizes other objects using the **Streamable = StreamIO::read_object()** function

Figure 1-3 Internalization control flow when object is not in a graph of related objects

A streamable object may be a node in a graph of related objects, that is, it may use the Relationship Service to connect to other objects and support the **CosStream::Node** interface. Such a streamable object simply delegates the

Streamable::internalize_from_stream() request back to the stream service, using the **StreamIO::write_graph()** operation.

The stream service then coordinates the externalization of the graph and calls the object back using the object's **CosStream::Node** interface.

Internalization Control Flow (streamable is a node)

Streamable object, recognizing that it is a node in a graph of related objects, delegates the internalization of the graph to the stream service using **StreamIO::read_graph(this_node)** operation.

StreamIO::read_graph, coordinates the internalization of the graph using **Node::internalize_node(this_sio)** operation.

Node reads its non-object data using the primitive **StreamIO::read...(data)** functions

Node read other objects using the **StreamIO::read_object(Streamable object)** function

Node reads its role objects using the **Role::internalize_role(this_sio)** operation.

StreamIO::read_graph reads the key for next node and uses the StreamableFactory interface to create the next node.

StreamIO object internalizes the traversal scoped identifiers for the externalized roles and relationships and internalizes the relationships using **Relationship::internalize()**.

Figure 1-4 Internalization control flow when object is in a graph of related objects

1.3 Object and Interface Hierarchies

This section identifies the objects required for the Externalization Service and important inheritance and use relationships that exist between their interfaces.

The Object Externalization Service can only externalize and internalize objects that inherit the **Streamable** interface. **Streamable** does not inherit any other interfaces. However, it must have an associated **StreamableFactory** that the Externalization Service implementation can find and use when internalizing the object.

Stream inherits the **LifeCycleObject** interface because clients of the Externalization Service need to remove these objects. The **StreamFactory** or **File StreamFactory** interfaces may be used to create stream objects.

In addition to the inheritance relationships described above, the class diagram in Figure 1-5 also shows the usage relationships between the service objects. **Stream externalize()** and **internalize()** operations invoke the **Streamable externalize_to_stream()** and **internalize_from_stream()** operations to write and read the appropriate object internal state. A **StreamIO** object is passed as an argument to these operations. The externalized object determines how much of its state must be put in the external representation, and can minimize saved state by recreating some state upon internalization. The **Streamable externalize_to_stream()** and **internalize_from_stream()** use **StreamIO** operations to actually put various data types and contained object references in the external representation. This allows **StreamIO** to put appropriate headers in the external representation so that the object can be recreated correctly during internalization. The **Stream** is responsible for providing an object that supports the **StreamIO** interface. The **Stream** object may support the **StreamIO** interface itself, or create another object that supports the **StreamIO** interface. The **Stream** and **StreamIO** implementations decide on the storage medium and data type representation conversion for different hardware, without requiring different implementation of the objects being externalized.

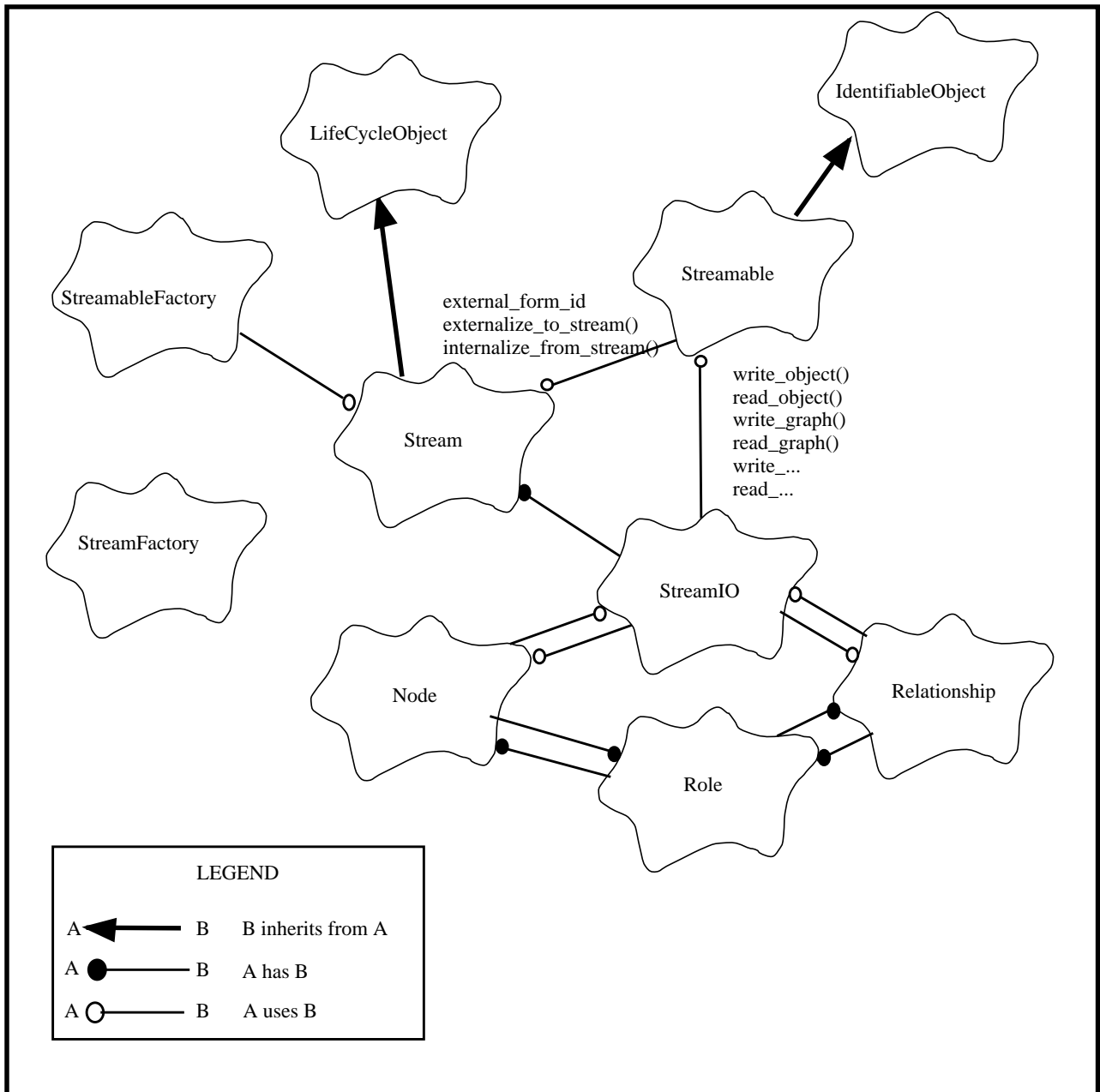


Figure 1-5 Object Externalization Service Booch Class (=Interface) Diagram

1.4 Interface Summary

The Externalization Service defines interfaces (using OMG IDL) to support the functionality described in the previous sections. The following tables give high level descriptions of the Externalization Service interfaces. Subsequent sections describe the interfaces in more detail.

Table 1-1 Client Functional Interfaces support client's model of externalization

Interface	Purpose	Primary Client
Stream	Holds external form of objects.	Clients that need to externalize and internalize objects.
StreamFactory	Creates and initializes stream objects.	Clients that need to create stream objects.
FileStreamFactory	Creates and initializes stream objects that stores data in a file.	Clients that need to create stream objects, and want the externalized data in a file.

Table 1-2 Service Construction Interfaces support service implementation's model of externalization

Interface	Purpose	Primary Client
Streamable	Provides its state to a stream for externalization, and gets its state from the stream on internalization.	The stream service implementation of externalization and internalization.
StreamableFactory	Creates and initializes streamable objects	The stream service internalization implementation.
StreamIO	Part of stream implementation that writes and reads object state to appropriately converted external form.	The externalizable objects that need to record and retrieve their state from a stream.

Table 1-3 Compound Externalization Interfaces support service implementation's model of graph externalization

Interface	Purpose	Primary Client
Node	Defines externalization and internalization operations on nodes in graphs of related objects.	The stream service implementation of externalization and internalization.
Relationship	Defines externalization and internalization operations on relationships.	The stream service implementation of externalization and internalization.
Role	Defines externalization and internalization operations on roles.	The stream service implementation of externalization and internalization.

1.4.1 Externalization Service Architecture: Audience/Bearer Mapping

Stream and **StreamFactory** are solely functional interfaces. Their audience is the client of the Externalization Service.

Streamable, **StreamableFactory**, and **StreamIO** are solely construction interfaces. The audience for **Streamable** is both the **Stream** and **StreamIO** objects. To be “externalizable,” objects must inherit the **Streamable** interface and provide implementations of its operations. The audience for **StreamIO** interface is the externalizable **Streamable** and **StreamableNode** objects. The **StreamIO** objects are part of the Externalization Service implementation.

The **Stream**, **StreamFactory**, and **StreamIO** objects are specific objects because their purpose is to provide a part of the Externalization Service. However, there may be many **Stream** and **StreamIO** instances in a system, since each represents a particular external representation of an object or group of objects.

Streamable and **StreamableFactory** objects are generic objects because their primary purpose is unrelated to the Externalization Service. Any definer or implementor of an object may choose to inherit the **Streamable** interface in order to support externalization/internalization of that object.

In summary:

- **Stream** and **StreamFactory** are specific functional interfaces
- **Streamable** and **StreamableFactory** are generic construction interfaces
- **StreamIO** is a specific construction interface

Contents

This chapter contains the following topics.

Topic	Page
“CosExternalization Module”	2-1
“CosStream Module”	2-5
“Specific Externalization Relationships”	2-15
“CosExternalizationContainment Module”	2-15
“CosExternalizationReference Module”	2-16
“Standard Stream Data Format”	2-18

2.1 CosExternalization Module

The client-functional interfaces defined by the the **CosExternalization** module are:

- **StreamFactory** interface, which creates a stream.
- **FileStreamFactory** interface, which has an operation that lets clients cause externalized data be stored in a file or internalize objects from a file they have been given.
- **Stream** interface, which can externalize one object or a group of objects; finalize the externalization, and internalize an object.

/File: CosExternalization.idl
//Part of the Externalization Service

```
#ifndef _COS_EXTERNALIZATION_IDL_  
#define _COS_EXTERNALIZATION_IDL_
```

```
#include <CosLifeCycle.idl>
#include <CosStream.idl>

#pragma prefix "omg.org"

module CosExternalization {
    exception InvalidFileNameError{};
    exception ContextAlreadyRegistered{};
    interface Stream: CosLifeCycle::LifecycleObject{
        void externalize(
            in CosStream::Streamable theObject);
        CosStream::Streamable internalize(
            in CosLifeCycle::FactoryFinder there)
            raises( CosLifeCycle::NoFactory,
                CosStream::StreamDataFormatError );
        void begin_context()
            raises( ContextAlreadyRegistered);
        void end_context();
        void flush();
    };
    interface StreamFactory {
        Stream create();
    };
    interface FileStreamFactory {
        Stream create(
            in string theFileName)
            raises( InvalidFileNameError );
    };
};
#endif /* ifndef _COS_EXTERNALIZATION_IDL_ */
```

2.1.1 *StreamFactory Interface*

2.1.1.1 *Creating a Stream Object*

Stream create();

Clients of the Object Externalization Service must create a **Stream** object before they can externalize or internalize any objects. Two factory interfaces are supported. The first, the **StreamFactory** interface has a **create()** operation that creates a stream without specifying any special characteristics of the implementation.

2.1.2 *FileStreamFactory Interface*

2.1.2.1 *Creating a Stream Object Associated with a File*

```
Stream create(  
  in string theFileName)  
  raises( InvalidFileNameError );
```

For clients that want to cause the externalized data stored in a file, or that need to internalize objects from a file they have been given, the **FileStreamFactory** interface has a **create()** operation that takes a string input parameter. The client sets this string to the filename of the file that will be used by the stream service to hold the external representation of the objects externalized, or that contains the external representation of objects that the client wishes to internalize. **Stream::externalize()** requests will append to any existing data in the file associated with a stream.

2.1.3 *Stream Interface*

2.1.3.1 *Externalizing an Object*

```
void externalize(  
  in CosStream::Streamable theObject);
```

Clients of the Object Externalization Service invoke **externalize()** on a **Stream** object passing the object reference of a **CosStream::Streamable** object, **theObject**, to be externalized. Only objects that are of type **CosStream::Streamable** can be externalized. Subsequently, clients invoke the **internalize()** operation on the **Stream** containing the external representation, and **Stream internalize()** operation creates a new object with state identical to what was externalized and returns the new object reference.

The implementation of **externalize()** writes implementation specific header information to the external representation it is maintaining, so that the correct object can be recreated at internalization time. This could be the factory key that was used to create the **CosStream::Streamable** object, or could include the interface type, implementation repository, or factory object names. The factory key may be obtained by from the **external_form_id** attribute of **theObject**. The **externalize()** implementation must then invoke the **CosStream::Streamable externalize_to_stream()** operation on **theObject** to cause the object's internal state to be written to the external representation. The **Stream** is responsible for providing an object that supports the **StreamIO** interfaces for the externalizable object to use in writing data to the stream service.

2.1.3.2 *Externalizing Groups of Objects*

```
void begin_context()  
  raises( ContextAlreadyRegistered);  
void end_context();
```

If a client wishes to externalize a set of objects with overlapping references and/or object relationships, the client invokes **begin_context()** on the **Stream**. This must be called before externalizing any of the set of objects, and **end_context()** must be called on the **Stream** after the entire set of objects has been externalized and before the **Stream** is used with another set of objects.

The **Stream** implementation establishes an association with the specified **Stream** object and a logical “context.” The **Stream** ensures that all objects externalized to this stream while this association lasts will be externalized in such a way that internalization will not create any duplicate objects. That is, the implementation of **Stream** checks for “context,” and for objects externalized in the same context handles overlapping or circular references and/or relationships between those objects. The association lasts until **end_context()** is called. The **Stream** raises the **ContextAlreadyRegistered** exception if **begin_context()** is called and a context is already established, perhaps through some other implementation dependent mechanism or perhaps because **end_context()** has not been called following a previous **begin_context()**.

2.1.3.3 *Completing Externalization*

void flush();

Clients invoke **flush()** to request that the external representation is committed to its final storage medium, whatever that may be. The implementation of **flush()** should attempt to ensure that the external representation is completely up-to-date in its final storage (e.g., memory buffer, file, tape, ...).

2.1.3.4 *Internalizing an Object*

**CosStream::Streamable internalize(
in CosLifeCycle::FactoryFinder there)
raises(CosLifeCycle::NoFactory,
CosStream::StreamDataFormatError);**

The implementation of **internalize()** must create an object with the correct interface and implementation to match the externalized representation and return a pointer to the new **CosStream::Streamable** object. The **internalize()** implementation must then invoke the **internalize_from_stream()** operation on the new object. The **CosStream::StreamDataFormatError** exception should be raised if an error is detected in the data format of the object header. The **CosLifeCycle::NoFactory** exception should be raised if the object cannot be created because an appropriate factory cannot be found. If the object cannot be created due to other reasons, an **ObjectCreationError** exception should be raised. Additional **CosStream::StreamDataFormat** exceptions may be raised by the **read_<type>** operations invoked by **internalize_from_stream()** operation due to errors in the externalized data format.

2.2 *CosStream Module*

The service construction interfaces defined by the **CosStream** module are:

- **Streamable** interface
- **StreamableFactory** interface
- **StreamIO** interface

If a **Streamable** object participates as a node in a graph of related objects, the *Streamable* object can delegate the externalization operation to the stream service. In particular, the **Streamable** object simply uses the **write_graph()** operation. The **write_graph()** operation expects a streamable object reference as a starting node. The stream service narrows the streamable object reference to **CosStream::Node**. The **write_graph()** then coordinates the orderly externalization of the graph of related objects. For more details on compound operations, see the Relationship Service specification and the Compound Life Cycle section in the Life Cycle Service specification.

The **CosStream** module defines interfaces for use by the **write_graph()** operation:

- **Node** interface
- **Role** interface
- **Relationship** interface
- **PropagationCriteriaFactory** interface

```
//File: CosStream.idl
//Part of the Externalization Service
// Modified from version 1.0 to include the previous CosCompoundExternalization
module

#ifndef _COS_STREAM_IDL_
#define _COS_STREAM_IDL_

#include <CosLifeCycle.idl>
#include <CosObjectIdentity.idl>
#include <CosGraphs.idl>

#pragma prefix "omg.org"

module CosStream {
    exception ObjectCreationError{};
    exception StreamDataFormatError{};

    interface StreamIO;
    interface Node;
    interface Role;
    interface Relationship;

    interface Streamable:
        CosObjectIdentity::IdentifiableObject {
            readonly attribute CosLifeCycle::Key external_form_id;
            void externalize_to_stream(
                in StreamIOtargetStreamIO);
            void internalize_from_stream(
```

```
        in StreamIOsourceStreamIO,
        in CosLifeCycle::FactoryFinder there)
        raises(CosLifeCycle::NoFactory,
               ObjectCreationError,
               StreamDataFormatError );
};

interface StreamableFactory {
    Streamable create_uninitialized();
};

interface StreamIO {
    void write_string(in string aString);
    void write_char(in char aChar);
    void write_octet(in octet anOctet);
    void write_unsigned_long(
        in unsigned long anUnsignedLong);
    void write_unsigned_short(
        in unsigned short anUnsignedShort);
    void write_long(in long aLong);
    void write_short(in short aShort);
    void write_float(in float aFloat);
    void write_double(in double aDouble);
    void write_boolean(in boolean aBoolean);
    void write_object(in Streamable aStreamable);
    void write_graph(in Node aNode);
    string read_string()
        raises(StreamDataFormatError);
    char read_char()
        raises(StreamDataFormatError );
    octet read_octet()
        raises(StreamDataFormatError );
    unsigned long read_unsigned_long()
        raises(StreamDataFormatError );
    unsigned short read_unsigned_short()
        raises( StreamDataFormatError );
    long read_long()
        raises(StreamDataFormatError );
    short read_short()
        raises(StreamDataFormatError );
    float read_float()
        raises(StreamDataFormatError );
    double read_double()
        raises(StreamDataFormatError );
    boolean read_boolean()
        raises(StreamDataFormatError );
    Streamable read_object(
        in CosLifeCycle::FactoryFinder there,
        in Streamable aStreamable)
        raises(StreamDataFormatError );
    void read_graph(
        in Node starting_node,
        in CosLifeCycle::FactoryFinder there)
        raises(StreamDataFormatError );
};
```

```

};

// the following are required for compound externalization

struct RelationshipHandle {
    CosRelationships::Relationship theRelationship;
    CosObjectIdentity::ObjectIdentifier constantRandomId;
};

interface Node : CosGraphs::Node, CosStream::Streamable{
    void externalize_node (in CosStream::StreamIO sio);
    void internalize_node (in CosStream::StreamIO sio,
        in CosLifeCycle::FactoryFinder there,
        out Roles rolesOfNode)
        raises ( CosLifeCycle::NoFactory);
};

interface Role : CosGraphs::Role {
    void externalize_role (in CosStream::StreamIO sio);
    void internalize_role (in CosStream::StreamIO sio);
    CosGraphs::PropagationValue externalize_propagation (
        in RelationshipHandle rel,
        in CosRelationships::RoleName toRoleName,
        out boolean sameForAll);
};

interface Relationship : CosRelationships::Relationship {
    void externalize_relationship (
        in CosStream::StreamIO sio);
    void internalize_relationship(
        in CosStream::StreamIO sio,
        in CosGraphs::NamedRoles newRoles);
    CosGraphs::PropagationValue externalize_propagation (
        in CosRelationships::RoleName fromRoleName,
        in CosRelationships::RoleName toRoleName,
        out boolean sameForAll);
};

interface PropagationCriteriaFactory {
    CosGraphs::TraversalCriteria create_for_externalize( );
};

};
#endif /* ifndef _COS_STREAM_IDL_ */

```

Since IDL only supports template instantiations rather than templates themselves, the fixed-point decimal template type cannot be used directly for the **write_fixed** and **read_fixed** operations. Instead, the **fixed** type instances must be passed to and from these routines as **any**s with **TypeCodes** of **tk_fixed**.

2.2.1 Standard Stream Data Format

The standard stream format for each new IDL type is shown in the table below. Also shown are the standard formats for types **char** and **string**, which have been extended to state explicitly that data is encoded as defined by ISO 8859-1.

Tag	CORBA Type	Data Format
x'F1'	char	one byte, encoded as defined by ISO 8859-1
x'FA'	string	null-terminated sequence of bytes, encoded as defined by ISO 8859-1
x'E1'	char	an unsigned long code set tag, followed by a one byte data value, encoded as defined by code set tag
x'E2'	string	an unsigned long code set tag, followed by a null-terminated sequence of characters, encoded as defined by code set tag
x'E3'	fixed<d,s>	an unsigned short byte count $(d+2)/2$, followed by $(d+2)/2$ bytes in CDR format.
x'FE'	wchar	an unsigned long code set tag, followed by a data value, encoded as defined by code set tag
x'FF'	wstring	an unsigned long code set tag, followed by a null-terminated sequence of wchar, encoded as defined by code set tag
x'FB'	long long	eight bytes, big-endian format
x'FC'	unsigned long long	eight bytes, big-endian format
x'FD'	long double	sixteen bytes, IEEE 754 format, sign bit in first byte

The first two entries in the table describe the current formats for **char** and **string**, modified only to state explicitly, rather than implicitly, that the encoding used is defined by ISO 8859-1. These existing formats are unchanged for backward compatibility purposes.

The next two entries (x'E1' and x'E2') define tagged formats for **char** and **string**, which consist of a code set tag (from the OSF Character and Code Set Registry) followed by an actual data value. The motivation for these tagged formats is to prevent information loss, which may occur for some native code sets when converted to ISO 8859-1 (i.e., when such data is externalized in the formats described in the first two entries). However, if character and string data is externalized in a form other than ISO 8859-1, some ORBs may not be able to internalize it successfully (e.g., because an appropriate converter is not available), thus reducing the portability of the externalized data. So, if maximum portability is desired, character and string data should be externalized in ISO 8859-1 form.

The remaining entries in the table describe the formats for the new IDL types. Note that the previous discussion about the tradeoff between portability and information loss for externalized character and string data also applies to wide character and wide string data. If maximum portability is desired, wide character and wide string data should be externalized in Unicode form, while if using this form would result in an unacceptable loss of information, then a form other than Unicode should be used.

Data values of type **wchar** and **wstring** are represented as one or more octets, or an unsigned integer, depending on the code set used. This is similar to the on-the-wire representation of **wchar** and **wstring** data.

2.2.2 The StreamIO Interface

The **write_<type>()** and **read_<type>()** operations on **StreamIO** are used by **Streamable externalize_to_stream()** and **internalize_from_stream()** operations to cause internal object state to be written to or read from the external representation. The **externalize_to_stream()** decomposes the internal state of an object in a series of primitive data type values that can be written and read with these operations. **StreamIO** supports writing and reading all the CORBA basic data types.

The implementation of the **write_...** and **read_...** operations are responsible for any desired conversion of the data and transferring the data to or from the desired external representation. Actual transfer of the representation to the final storage medium may be deferred until the **flush()** operation. All details of the external representation format, storage medium, and buffering are specific to the implementation. Different implementations may support buffering of the external representation data in memory, converting data values to a canonical binary form for exchange across big/little endian CPU hardware, conversion of data to a canonical text form for readability or to facilitate mailing objects across networks, use of various storage mediums such as memory, filesystem, tape or other differences. See Section 2.2.1, “Standard Stream Data Format,” on page 2-8 for information on a portable external representation. A **StreamDataFormatError** exception should be raised if errors are detected in the data format of the external representation.

In support of integrating the Externalization Service with the Transaction and Persistent Object Services, the **read_object** operation supports the internalization to existing objects. The semantics of the operation are that if the **aStreamable** parameter is Null, then the **FactoryFinder** parameter is used to create an instance for internalize. If the **aStreamable** parameter is not Null, then the **StreamIO** implementation will internalize to a streamable object. This semantic allows the Externalization Service to be used as a Persistent Object Service protocol and to support the restore operation on existing objects in the case of an aborted transaction.

2.2.3 The Streamable Interface

Object implementors must inherit from the **Streamable** interface if they want an object to be externalizable. Three operations must be implemented.

Comparing Streamable Objects

```
boolean CosObjectIdentity::IdentifiableObject::is_identical(
    in CosObjectIdentity::IdentifiableObject anObject);
readonly unsigned long constant_random_id;
```

A **Streamable** object inherits from **CosObjectIdentity::IdentifiableObject**, and therefore must support a **constant_random_id** attribute and an **is_identical()** operation. The stream service uses these to compare objects when detecting cycles or overlapping references in objects being externalized to the same stream in the same context or within the same graph. The **constant_random_id** attribute value does not have to be unique, but a unique value may substantially speed up the externalization process.

Creation Key for a Streamable Object

readonly attribute CosLifecycle::Key external_form_id;

An **Streamable** object must support a readonly attribute, **external_form_id**, which is a key that can be given to a factory finder in order to find a factory that could have created this object. The stream service may use this attribute during internalization to create an object that can reinitialize itself from the externalized data.

Writing the Object's State to a Stream

**void externalize_to_stream(
in StreamIOtargetStreamIO);**

The **externalize_to_stream()** operation is responsible for decomposing an externalizable object's internal state into a series of primitive data type values and object references. The **externalize_to_stream()** function must write out all the necessary primitive data values using the **write_<type>()** operations on the **targetStreamIO** for non-object data types. If this object has other object references, then, normally, those objects should also be written out using the **write_object()** operation on the **targetStreamIO**. However, it is up to the **Streamable** implementor to decide which referenced objects should be externalized with this object. The primitive data values must all be written before any of the embedded objects references are written.

If the **Streamable** is a node in a graph, then it should delegate the **externalize_to_stream()** to the **StreamIO** by invoking **write_graph()**. The object would subsequently receive an **externalize_node_to_stream()** and write out its internal state as described above. **Node** objects should not call **write_object()** for other nodes in their graph, but may call **write_object()** for object references that are not for nodes in their graph.

2.2.3.1 Reinitializing the Object's State from a Stream

**void internalize_from_stream(
in StreamIOsourceStreamIO,
in CosLifecycle::FactoryFinder there)
raises(CosLifecycle::NoFactory,
ObjectCreationError,
StreamDataFormatError);**

The **internalize_from_stream()** operation is responsible for reinitializing the object's internal state from the series of primitive data type values and object references that are written/flattened during **externalize_to_stream()**. The **internalize_from_stream()** operation should read in all the necessary internal state of the object using the **read_<type>()** operations on the **sourceStreamIO** for non-object data types. If this object has other object references that were externalized using **write_object()**, then those objects should be recreated using the **read_object()** operation on the **sourceStreamIO** with the same **FactoryFinder** argument as the **there** parameter passed in to the **internalize_from_stream()** operation. The **read_<type>()** and **read_object()** operations for the various portions of the object's internal state must be invoked in the same order in which they are written by the **externalize_to_stream()** implementation. The **internalize_from_stream()** must also initialize any additional state that was not externalized because it can be derived from other state information. Therefore, the **externalize_to_stream()** and **internalize_from_stream()** operations must be designed to complement each other.

If the **Streamable** is a node in a graph, then it should delegate the **internalize_to_stream()** to the **sourceStreamIO** by invoking **read_graph()** with the same **FactoryFinder** argument as the **there** parameter passed in to the **internalize_from_stream()** operation. The **Streamable** (also **Node**) object would subsequently receive an **internalize_node_to_stream()** and read in its internal state as described above. **Node** objects should not call **read_object()** for other nodes in their graph, but may call **read_object()** for object references that are not for nodes in their graph..

The **ObjectCreationError** and **StreamDataFormatError** exceptions originate from the **read_object()** and **read_<type>** operations on the **sourceStreamIO**, and are not explicitly raised by the **internalize_from_stream()** code.

2.2.4 The StreamableFactory Interface

2.2.4.1 Creating a Streamable Object

Streamable create_uninitialized();

The stream service must be able to create a **Streamable** object in order to internalize an object from the stream's externalized data. For any externalizable object, a **StreamableFactory** object must exist that supports creation of that object. This factory must be findable using the **readonly external_form_id Key attribute** of the streamable object. The stream service implementation could store this key during externalization and use it during internalization to find the factory that can create the externalized object. However, a stream implementation may use other means to create the object during internalization. The **create_uninitialized()** operation on the **StreamableFactory** should create the associated streamable object. This streamable object does not have to be initialized, since that can be done on the subsequent **internalize_from_stream()** operation on the newly created streamable object.

2.2.5 The Node Interface

The **Node** interface defines operations to internalize and externalize a node.

2.2.5.1 Externalizing a Node

```
void externalize_node (in CosStream::StreamIO sio);
```

The **externalize_node()** operation transfers the node's state to the stream given by the **sio** parameter. The node is responsible to externalize its roles as well. The node can accomplish this by writing the role's key to the stream and using the **Role::externalize_role()** operation.

2.2.5.2 Internalizing a Node

```
void internalize_node (in CosStream::StreamIO sio,  
in CosLifeCycle::FactoryFinder there,  
out Roles rolesOfNode)  
raises ( CosLifeCycle::NoFactory);
```

The **internalize_node()** operation causes a node and its roles to be internalized from the stream **sio**.

It is the node's responsibility to create and internalize its roles. It can do this by reading the key for a role from the stream and using the **CosStream::StreamableFactory** interface to create the uninitialized role and the **CosStream::internalize_role()** operation to internalize the role. The new roles should be collocated with the factory finder given by the **there** parameter.

The result of a **internalize_node()** operation is a sequence of roles.

Figure 2-1 illustrates the result of an internalize. A node, when it is born, is not in any relationships with other objects. That is, the roles in the new node are "disconnected." It is the **read_graph()** operation's job to correctly establish new relationships.



Figure 2-1 Internalizing a node returns the new object and the corresponding roles.

If an appropriate factory to internalize the roles cannot be found, the **NoFactory** exception is raised. The exception value indicates the key used to find the factory.

In addition to the **NoFactory** exception, implementations may raise standard CORBA exceptions. For example, if resources cannot be acquired for the internalized node, **NO_RESOURCES** will be raised.

2.2.6 The Role Interface

The **Role** interface defines operations to externalize and internalize a role. The **Role** interface also defines an operation to return the propagation value for the externalize operation.

The implementation of a **CosStream::Node** operation can call these operations on roles. For example, an implementation of **externalize** on a node can call the **externalize** operation on the **Role**.

2.2.6.1 Externalizing a Role

```
void externalize_role (in CosStream::StreamIO sio);
```

The **externalize_role()** operation transfers the role's state to the stream **sio**.

2.2.6.2 Internalizing a Role

```
void internalize_role (in CosStream::StreamIO sio);
```

The **internalize_role()** operation causes a role to read its state from the stream given by **sio**.

2.2.6.3 Getting a Propagation Value

```
CosGraphs::PropagationValue externalize_propagation (  
in RelationshipHandle rel,  
in CosRelationships::RoleName toRoleName,  
out boolean sameForAll);
```

The **externalize_propagation()** operation returns the propagation value to the role **toRoleName** for the externalization operation and the relationship **rel**. If the role can guarantee that the propagation value is the same for all relationships in which it participates, **sameForAll** is true.

2.2.7 The Relationship Interface

The **Relationship** interface defines operations to externalize and internalize a relationship. The **Relationship** interface also defines an operation to return the propagation values for the externalize operations.

2.2.7.1 Externalizing the Relationship

```
void externalize_relationship (  
in CosStream::StreamIO sio);
```

The **externalize_relationship()** operation transfers the role's state to the stream **sio**.

2.2.7.2 *Internalizing the Relationship*

```
void internalize_relationship(  
    in CosStream::StreamIO sio,  
    in CosGraphs::NamedRoles newRoles);
```

The **internalize_relationship()** operation internalizes the state of a relationship from the stream given by **sio**.

The values of the internalized relationship's attributes are defined by the implementation of this operation. However, the **named_roles** attribute of the newly created relationship must match **newRoles**. That is, the internalized relationship relates objects represented by **newRoles** parameter, not the by the original relationship's named roles.

2.2.7.3 *Getting a Propagation Value*

```
CosGraphs::PropagationValue externalize_propagation (  
    in CosRelationships::RoleName fromRoleName,  
    in CosRelationships::RoleName toRoleName,  
    out boolean sameForAll);
```

The **propagation_for()** operation returns the relationship's propagation value from the role **fromRoleName** to the role **toRoleName** for the externalization operation. If the role named by **fromRoleName** can guarantee that the propagation value is the same for all relationships in which it participates, **sameForAll** is true.

2.2.8 *The PropagationCriteriaFactory Interface*

The **CosGraphs** module in the Relationship Service defines a general service for traversing a graph of related objects. The service accepts a "call-back" object supporting the **CosGraphs::TraversalCriteria** interface. Given a node, this object defines which edges to emit and which nodes to visit next.

The **PropagationCriteriaFactory** creates a **TraversalCriteria** object that determines which edges to emit and which nodes to visit based on propagation values for the compound externalization operations.

2.2.8.1 *Create a Traversal Criteria Based on Externalization Propagation*

```
CosGraphs::TraversalCriteria create_for_externalize( );
```

The **create_for_externalize** operation returns a **TraversalCriteria** object for an operation op that determines which edges to emit and which nodes to visit based on propagation values for op. For a more detailed discussion see the Relationship Service specification.

2.3 Specific Externalization Relationships

The Relationship Service defines two important relationships: containment and reference. Containment is a one-to-many relationship. A container can contain many containees; a containee is contained by one container. Reference, on the other hand, is a many-to-many relationship. An object can reference many objects; an object can be referenced by many objects.

Containment is represented by a relationship with two roles: the **ContainsRole**, and the **ContainedInRole**. Similarly, reference is represented by a relationship with two roles: **ReferencesRole** and **ReferencedByRole**.

Compound externalization adds externalization semantics to these specific relationships. That is, it defines propagation values for containment and reference.

2.4 CosExternalizationContainment Module

The **CosExternalizationContainment** module defines the following interfaces:

- **Relationship** interface
- **ContainsRole** interface
- **ContainedInRole** interface

```
//File: CosExternalizationContainment.idl
//Part of the Externalization Service
// modified from version 1.0 to use CosStream module
// instead of CosCompoundExternalization

#ifndef _COS_EXTERNALIZATION_CONTAINMENT_IDL_
#define _COS_EXTERNALIZATION_CONTAINMENT_IDL_

#include <CosContainment.idl>
#include <CosStream.idl>

#pragma prefix "omg.org"

module CosExternalizationContainment {

    interface Relationship :
        CosStream::Relationship,
        CosContainment::Relationship {};

    interface ContainsRole :
        CosStream::Role,
        CosContainment::ContainsRole {};

    interface ContainedInRole :
        CosStream::Role,
        CosContainment::ContainedInRole {};

};
#endif /* ifndef _COS_EXTERNALIZATION_CONTAINMENT_IDL_*/
```

The **CosExternalizationContainment** module does not define new operations. It merely “mixes in” interfaces from the **CosStream** and **CosContainment** modules. Although it does not add any new operations, it refines the semantics of these operations:

The **CosExternalizationContainment::ContainsRole::propagation_for** operation returns the following:

operation	ContainsRole to ContainedInRole
externalize	deep

The **CosExternalizationContainment::ContainedInRole::propagation_for()** operation returns the following::

operation	ContainedInRole to ContainsRole
externalize	none

The **CosRelationships::RoleFactory::create_role()** operation will raise the **RelatedObjectTypeError** if the related object passed as a parameter does not support the **CosStream::Node** interface.

The **CosRelationships::RelationshipFactory::create()** operation will raise **DegreeError** if the number of roles passed as arguments is not 2. It will raise **RoleTypeError** if the roles are not **CosExternalizationContainment::ContainsRole** and **CosExternalizationContainment::ContainedInRole**. It will raise **MaxCardinalityExceeded** if the **CosExternalizationContainment::ContainedInRole** is already participating in a relationship.

2.5 *CosExternalizationReference Module*

The **CosExternalizationReference** module defines these interfaces:

- **Relationship** interface
- **ReferencesRole** interface
- **ReferencedByRole** interface

```
//File: CosExternalizationReference.idl
//Part of the Externalization Service
// modified from version 1.0 to use CosStream module
// instead of CosCompoundExternalization

#ifndef _COS_EXTERNALIZATION_REFERENCE_IDL_
#define _COS_EXTERNALIZATION_REFERENCE_IDL_
```



```

#include <CosReference.idl>
#include <CosStream.idl>

#pragma prefix "omg.org"

module CosExternalizationReference {

    interface Relationship :
        CosStream::Relationship,
        CosReference::Relationship {};

    interface ReferencesRole :
        CosStream::Role,
        CosReference::ReferencesRole {};

    interface ReferencedByRole :
        CosStream::Role,
        CosReference::ReferencedByRole {};
};
#endif /* ifndef _COS_EXTERNALIZATION_REFERENCE_IDL_ */

```

The **CosExternalizationReference** module does not define new operations. It merely “mixes in” interfaces from the **CosStream** and **CosReference** modules. Although it does not add any new operations, it refines the semantics of these operations:

The **CosExternalizationReference::ReferencesRole::propagation_for()** operation returns the following:

operation	ReferencesRole to ReferencedByRole
externalize	none

The **CosExternalizationReference::ReferencedByRole::propagation_for()** operation returns the following::

operation	ReferencedByRole to ReferencesRole
externalize	none

The **CosRelationships::RoleFactory::create_role()** operation will raise the **RelatedObjectTypeError** if the related object passed as a parameter does not support the **CosStream::Node** interface.

The **CosRelationships::RelationshipFactory::create()** operation will raise **DegreeError** if the number of roles passed as arguments is not 2. It will raise **RoleTypeError** if the roles are not **CosExternalizationReference::ReferencesRole** and **CosExternalizationReference::ReferencedByRole**.

2.6 Standard Stream Data Format

An externalization client may create a stream that supports a specific external representation data format that is intended to be portable across different CORBA implementations and on different CPU hardware. A client creates such a **Stream** object using a factory found by specifying a Key whose only **NameComponent** has an **NameComponent::id** whose value is the string literal “StandardExternalizationFormat”.

That format is described in this section.

2.6.1 OMG Externalized Object Data

1 byte

tag byte = x'F0'	Key info	Object info
------------------	----------	-------------

A leading “tag” byte with a value of x'F0” marks the beginning of an object’s externalized data. Following this is data associated with a Key that can be used to internalize the object. The key information is then followed by the data written to the **StreamIO** for the object’s state.

Key Info

1 byte

length = i	1st id string	2nd id string	...	i'th id string
------------	---------------	---------------	-----	----------------

The key information consists of a byte containing an integer value, “i”, that indicates how many **CosNaming::NameComponents** make up the associated Key.

This byte is followed by “i” null-terminated sequences of char values that represent the **CosNaming::NameComponent::id** values for the Key. These values correspond to the C mapping of a CORBA string type. The **NameComponent::kind** values are not stored in this external data format.

Object Info

1 byte

1 byte

tag byte	data value	tag byte	data value	...
----------	------------	----------	------------	-----

The object information is the sequence of bytes generated for one or more **write_<type>** operation. For each **write_<type>** operation, a single “tag” byte identifying the type of the primitive data is followed by the data. The tag byte gives the

internalization implementation enough information to skip past object state for objects that cannot be created, for example when a compatible implementation cannot be found on the internalizing ORB.

The tag byte values, and data formats for each type are as indicated below for basic CORBA data types:

Table 2-1 CORBA Tag Byte Values and Data Formats

tag	CORBA type	data format
x'F1'	Char	one byte
x'F2'	Octet	one byte
x'F3'	Unsigned Long	four bytes, big-endian format
x'F4'	Unsigned Short	two bytes, big-endian format
x'F5'	Long	four bytes, big-endian format
x'F6'	Short	two bytes, big-endian format
x'F7'	Float	four bytes, IEEE 754 single precision format, sign bit in first byte
x'F8'	Double	eight bytes, IEEE 754 double precision format, sign bit first byte
x'F9'	Boolean	TRUE=>one byte==1, FALSE=>one byte==0
x'FA'	String	null-terminated sequence of bytes

2.6.2 Externalized Repeated Reference Data

1	4	(bytes)
x'04'	Object number	

This format is used only when multiple objects reference the same object. Instead of storing the referenced object multiple times, the duplicate reference objects are stored in this format. Note that the object is represented by a long object number which indicates that the object has been stored already.

2.6.3 Externalized NIL Data

1 (byte)

x'05'

This is a special format used to indicate that there is no object stored in the stream.

References

A

1. James Rumbaugh, "Controlling Propagation of Operations using Attributes on Relations." *OOPSLA 1988 Proceedings*, pg. 285-296
2. James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy and William Lorensen, "Object-oriented Modeling and Design." Prentice Hall, 1991.
3. Grady Booch, "Object Oriented Design with Applications." The Benjamin/Cummings Publishing Company, Inc., 1991.

