

Externalization Service Specification 8

8.1 Service Description

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.

8.2 Service Structure

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

8.2.1 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.

8.2.2 *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 an *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 8-2

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 8-1 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 *CosCompoundExternalization::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 *CosCompoundExternalization::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 8-2 Externalization control flow when streamable object is a node in a graph of related objects


8.2.4 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 8-4

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 8-3 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 *CosCompoundExternalization::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 *CosCompoundExternalization::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 8-4 Internalization control flow when object is in a graph of related objects

8.3 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 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.

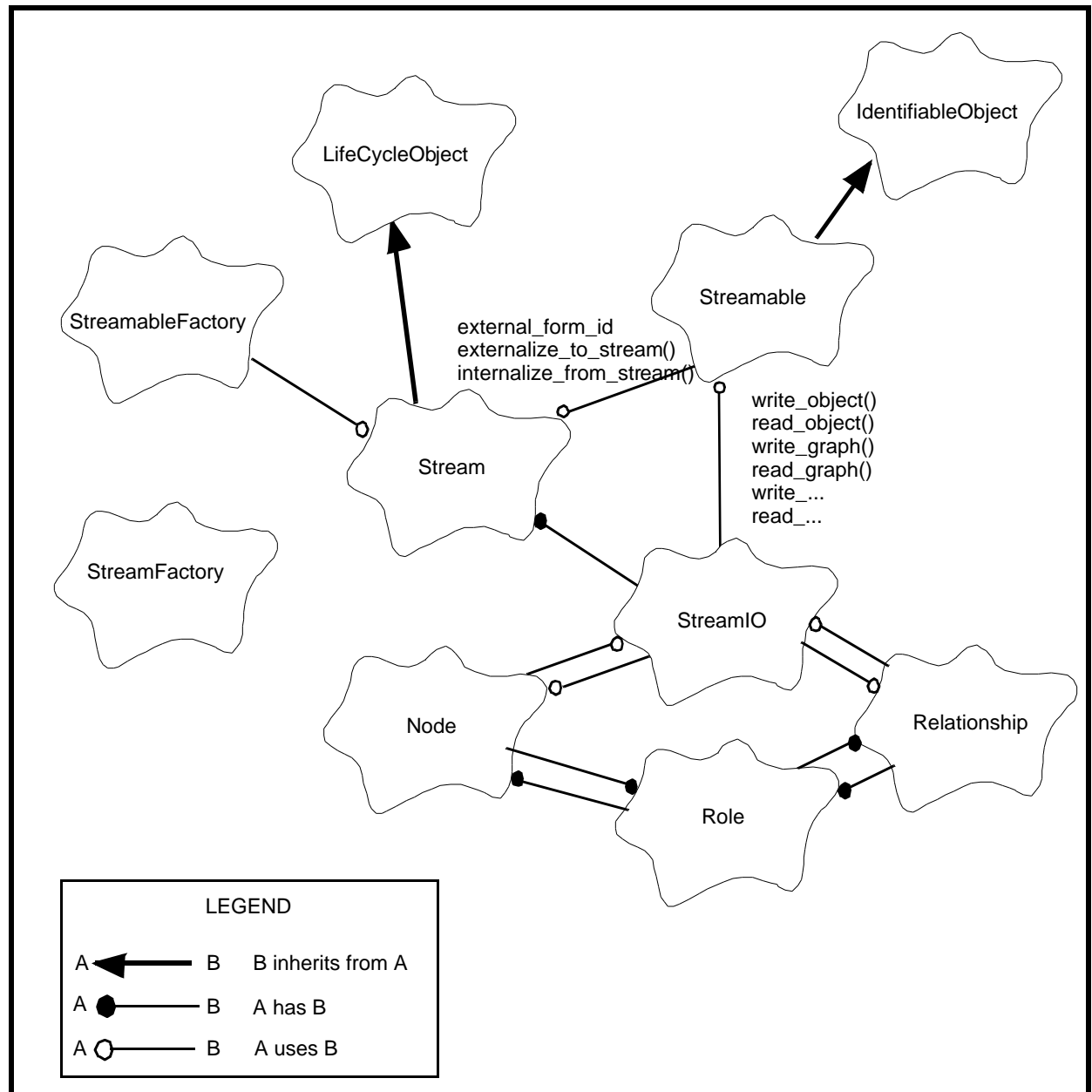


Figure 8-5 Object Externalization Service Booch Class (=Interface) Diagram

8.4 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.

Figure 8-6 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.

Figure 8-7 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.

Figure 8-8 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.

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

8.5 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.

```
#include <LifeCycle.idl>
#include <Stream.idl>
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 );
    };
};
```

8.5.1 StreamFactory Interface

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.

8.5.2 *FileStreamFactory Interface*

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.

8.5.3 *Stream Interface*

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.

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()`.

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, ...).

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.

8.6 *CosStream* Module

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

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

```
#include <LifeCycle.idl>
#include <ObjectIdentity.idl>
#include <CompoundExternalization.idl>
module CosStream {
    exception ObjectCreationError{};
    exception StreamDataFormatError{};
    interface StreamIO;

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

Figure 8-9 The *CosStream* module

```

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 CosCompoundExternalization::Node);
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 FactoryFinder there,
    in Streamable aStreamable)
    raises(StreamDataFormatError );
void read_graph(
    in CosCompoundExternalization::Node
starting_node,
    in FactoryFinder there)
    raises(StreamDataFormatError );
};

```

Figure 8-9 The CosStream module

8.6.1 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 the Standard Stream Data Format section 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 streamable parameter is Null, then the `FactoryFinder` parameter is used to create an instance for internalize. If the streamable parameter is not Null, then the `StreamIO` implementation will internalize to the a streamable object. This semantic allows the Externalization Service to be used as a Persistent Object Service protocol and support the restore operation on existing objects in the case of an aborted transaction.

8.6.2 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 StreamIO targetStreamIO);
```

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.

Reinitializing the Object's State from a Stream

```
void internalize_from_stream(
    in StreamIO sourceStreamIO,
    in FactoryFinder there);
```

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.

8.6.3 The *StreamableFactory* Interface

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.

8.7 *CosCompound Externalization Module*

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

CosCompoundExternalization::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 *CosCompoundExternalization* module defines the *Node*, *Role*, *Relationship* and *PropagationCriteriaFactory* interfaces for use by the `write_graph()` operation.

The *CosCompoundExternalization* module is shown in Figure 8-10. Detailed descriptions of the interfaces follow.

```
#include <Graphs.idl>
#include <Stream.idl>

module CosCompoundExternalization {
    interface Node;
    interface Role;
    interface Relationship;
    interface PropagationCriteriaFactory;

    struct RelationshipHandle {
        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);
    };
};
```

Figure 8-10 The *CosCompoundExternalization* Module

```

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( );
    };
};

```

Figure 8-10 The CosCompoundExternalization Module (*Continued*)

8.7.1 The Node Interface

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

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.

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 *CosCompoundExternalization::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 8-11 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 8-11 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.

8.7.2 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 *CompoundExternalization::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*.

Externalizing a Role

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

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

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*.

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.

8.7.3 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.

Externalizing the Relationship

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

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

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.

Getting a Propagation Value

```
::CosGraphs::PropagationValue externalize_propagation (
    in::CosRelationships::RoleName fromRoleName,
    in::CosRelationship::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.

8.7.4 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.

Create a Traversal Criteria Based on Externalization Propagation

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

The `create` 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 chapter, section 9.4.2.

8.8 *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.

8.9 The *CosExternalizationContainment* Module

The *CosExternalizationContainment* module defines the following interfaces:

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

```
#include <Containment.idl>
#include <CompoundExternalization.idl>

module CosExternalizationContainment {

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

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

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

};
```

Figure 8-12 The *CosExternalizationContainment* module

The *CosExternalizationContainment* module does not define new operations. It merely “mixes in” interfaces from the *CosCompoundExternalization* 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 *CosCompoundExternalization::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.

8.10 The *CosExternalizationReference* Module

The *CosExternalizationReference* module defines these interfaces:

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

```
#include <Reference.idl>
#include <CompoundExternalization.idl>

module CosExternalizationReference {

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

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

    interface ReferencedByRole :
        ::CosCompoundExternalization::Role,
        ::CosReference::ReferencedByRole {};

};
```

Figure 8-13 The *CosExternalizationReference* module

The *CosExternalizationReference* module does not define new operations. It merely “mixes in” interfaces from the *CosCompoundExternalization* 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 *CosCompoundExternalization::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*.

8.11 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.

8.11.1 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 *Naming::NameComponent*’s make up the associated Key.

This byte is followed by “i” null-terminated sequences of char values that represent the *Naming::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 8-1 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

8.11.2 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.

8.11.3 Externalized NIL Data

1 (byte)

x'05'

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

8.12 References

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.

