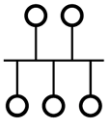

Orbix®

Persistent State Service Reference

Version 6.1, December 2003

Making Software Work Together™

Updated: 01-Dec-2003

M 3 1 7 3

Contents

CosPersistentState Overview

The persistent state service (PSS) is a CORBA-friendly object-oriented database. PSS storage objects can hold any kind of IDL type. The Orbix implementation of PSS is organized into three modules and an object factory class:

- "CosPersistentState Overview"

  The `CosPersistentState` module is the standard OMG service for persistent objects.
- "IT_PSS Overview"

  The `IT_PSS` module provides various proprietary useful features such as queries.
- The IT_PSS_DB Module Overview

  The Orbix implementation of PSS is targeted at relational and relational-like database back-ends. It is not restricted to any particular database system.

The CosPersistentState module's features are listed in Table 1:

Table 1: *The CosPersistentState Module*

Common Data Types	Interfaces
AccessMode Type ForUpdate Enumeration IsolationLevel Type NotFound Exception Parameter Structure ParameterList Sequence Pid Type ShortPid Type TransactionalSessionList Sequence TypeId Type YieldRef Enumeration	CatalogBase Connector EndOfAssociationCallback Session StorageHomeBase TransactionalSession **Native Types and Helper Classes** CosPersistentState_Factory StorageHomeFactory StorageObjectBase StorageObjectFactory StorageObjectRef

The rest of this chapter describes the common data types for the module.

CosPersistentState::AccessMode Type

```
// PSDL Code
typedef short AccessMode;
```

The mode of access for a storage object. Valid values include:

```
READ_ONLY
READ_WRITE
```

The AccessMode READ_WRITE is higher than READ_ONLY.

CosPersistentState::ForUpdate Enumeration

```
// PSDL Code
enum ForUpdate { FOR_UPDATE };
```

Used in the language mapping to define an overloaded accessor method that can update the state member.

Examples

For example, a state member whose type is an abstract storagetype is mapped to a read-only accessor, a read-write (update) accessor, and a modifier:

```
// PSDL
abstract storagetype A {};
abstract storagetype B {
    state A embedded;
};
```

This PSDL code maps to:

```
// C++
class B : public virtual StorageObject {
public:
    virtual const A& embedded() const = 0;
    virtual A& embedded(CosPersistentState::ForUpdate) = 0;
    virtual void embedded(const A&) = 0; // copies
};
```

CosPersistentState::IsolationLevel Type

```
// PSDL Code
typedef short IsolationLevel;
const IsolationLevel READ_UNCOMMITTED  = 0;
const IsolationLevel READ_COMMITTED    = 1;
const IsolationLevel REPEATABLE_READ   = 2;
const IsolationLevel SERIALIZABLE      = 3;
```

When data is accessed through a transactional session actively associated with a resource, undesirable phenomena such as dirty reads or non-repeatable reads may occur. An isolation level controls user access to these kinds of phenomenon during a transactional session.

Valid `IsolationLevel` values include the following:

`READ_UNCOMMITTED` — When a resource has this isolation level, its user may experience the dirty reads and the non-repeatable reads phenomena.

`READ_COMMITTED` — When a resource has this isolation level, its user may experience the non-repeatable reads phenomenon, but not the dirty reads phenomenon.

SERIALIZABLE	When a resource has this isolation level, its user is protected from both the dirty reads and the non-repeatable reads phenomena
REPEATABLE_READ	This isolation level is reserved for future use.

A dirty read occurs when a resource is used to read the uncommitted state of a storage object. For example, suppose a storage object is updated using resource 1. The updated storage object's state is read using resource 2 before resource 1 is committed. If resource 1 is rolled back, the data read with resource 2 is considered never to have existed.

A non-repeatable read occurs when a resource is used to read the same data twice but different data is returned by each read. For example, suppose resource 1 is used to read the state of a storage object. Resource 2 is used to update the state of this storage object and resource 2 is committed. If resource 1 is used to reread the storage object's state, different data is returned.

See Also CosPersistentState::TransactionalSession

CosPersistentState::NotFound Exception

```
// PSDL Code
exception NotFound {};
```

An exception that indicates that a storage object or registry connector cannot be found.

CosPersistentState::Parameter Structure

```
// PSDL Code
struct Parameter {
    string name;
    any val;
};
```

A parameter in a list of parameters when creating a session.

Parameters

`name`	The parameter's name.
`val`	The value in the parameter.

See Also `CosPersistentState::ParameterList`
`CosPersistentState::Connector::create_basic_session()`
`CosPersistentState::Connector::create_transactional_session()`

CosPersistentState::ParameterList Sequence

```
// PSDL Code
typedef sequence<Parameter> ParameterList;
```

A sequence of `Parameter` structures.

See Also `CosPersistentState::Parameter`

CosPersistentState::Pid Type

```
// PSDL Code
typedef CORBA::OctetSeq Pid;
```

A global persistent object identifier that storage objects use. The scope of the `Pid` is all storage objects that can be accessed through the same catalog.

See Also `CosPersistentState::ShortPid`

CosPersistentState::ShortPid Type

```
// PSDL Code
typedef CORBA::OctetSeq ShortPid;
```

A storage object identifier that is unique within a storage home family.

See Also `CosPersistentState::Pid`

CosPersistentState::TransactionalSessionList Sequence

```
// PSDL Code
```

```
typedef sequence<TransactionalSession> TransactionalSessionList;
```

A list of transactional sessions.

See Also CosPersistentState::TransactionalSession
CosPersistentState::Connector::sessions()

CosPersistentState::TypeId Type

```
// PSDL Code
typedef string TypeId;
```

A string that identifies a PSDL type. The format of a PSDL type id is the same as the IDL format of repository ids, except that the prefix is `PSDL`, not `IDL`.

See Also CORBA::RepositoryId
CosPersistentState::Connector

CosPersistentState::YieldRef Enumeration

```
// PSDL Code
enum YieldRef { YIELD_REF };
```

Used in the language mapping to define overloaded methods that yield incarnations and references as parameters.

Examples For example, a state member whose type is a reference to an abstract storagetype is mapped to two accessors and two modifier methods:

```
// PSDL
abstract storagetype Bank;
abstract storagetype Account {
    state ref<Bank> my_bank;
};
```

The mapping shows that one of the accessor methods takes no parameter and returns a storage object incarnation, and the other takes a `YieldRef` parameter and returns a reference:

```
// C++
class Account : public virtual StorageObject {
public:
    virtual Bank* my_bank() const= 0;
    virtual const BankRef* my_bank(
```

```
            CosPersistentState::Yield-Ref yr
        ) const = 0;

        virtual void my_bank(Bank* b) = 0;
        virtual void my_bank(const BankRef* b) = 0;
    };
```

CosPersistentState::CatalogBase Interface

The CatalogBase interface is the base interface for the implementation of a local catalog object.

```
// PSDL in module CosPersistentState
local interface CatalogBase {

    readonly attribute AccessMode access_mode;

    StorageHomeBase find_storage_home(
        in string storage_home_type_id
    )
        raises (NotFound);

    StorageObjectBase find_by_pid(
        in Pid the_pid
    )
        raises (NotFound);

    void flush();
    void refresh();
    void free_all();
    void close();
};
```

CatalogBase::access_mode Attribute

```
// PSDL code
readonly attribute AccessMode access_mode;
```

Returns the access mode of this catalog. When the access mode is READ_ONLY, the storage object incarnations obtained through storage home instances provided by this catalog are read-only.

CatalogBase::close()

```
// PSDL code
void close();
```

Terminates the catalog. If the catalog is associated with one or more transactions when `close()` is called, these transactions are marked as roll-back only. When closed, the catalog is also flushed for a non-transactional session.

CatalogBase::find_by_pid()

```
// PSDL code
StorageObjectBase find_by_pid(
    in Pid the_pid
)
    raises (NotFound);
```

Attempts to locate a storage object and returns an incarnation of it.

Parameters

`the_pid`	The operation uses the given `Pid` to find the storage object in the storage homes provided by the target catalog.

Exceptions

`NotFound exception`	The operation cannot find a storage object with this `Pid`.

CatalogBase::find_storage_home()

```
// PSDL code
StorageHomeBase find_storage_home(
    in string storage_home_type_id
)
    raises (NotFound);
```

Returns a storage home instance.

Parameters

`storage_home_type_id`	The operation looks up a PSDL-defined storage home with this Id in the catalog's default data-store. The format of this parameter is mostly implementation-defined. In the case of type-specific catalogs (declared in PSDL), the provided declarations define valid values for this parameter. The operation can also interpret Ids that have the form of a PSDL type Id. For example: `PSDL:com/acme/PersonStoreImpl:1.0`

Exceptions

`NotFound`	The operation cannot find a storage home that matches the given storage home Id.

CatalogBase::flush()

```
// PSDL code
void flush();
```

Writes to disk any cached modifications of storage object incarnations managed by this catalog. PSS can cache some *dirty* data, thus, when an application creates a new storage object or updates a storage object, the modification is not written directly to disk.

CatalogBase::free_all()

```
// PSDL code
void free_all();
```

Instructs the catalog implementation to set the reference count of all its PSDL storage object instances to 0.

CatalogBase::refresh()

```
// PSDL code
void refresh();
```

Refreshes any cached storage object incarnations accessed (read) by this catalog. In addition to caching write data, PSS can cache data read from datastores.

Note: This operation can invalidate any direct reference to a storage object incarnation's data member. Most applications do not use `refresh()`, so calling it is unusual.

CosPersistentState::Connector Interface

A connector is a local object that represents a given PSS implementation. Sessions are created by connectors. You obtain a connector of a given ORB by calling CORBA::ORB::resolve_initial_references() with the ObjectId of PSS.

```
// PSDL code in module CosPersistentState
local interface Connector {

    readonly attribute string implementation_id;

    Pid get_pid(
        in StorageObjectBase obj
    );

    ShortPid get_short_pid(
        in StorageObjectBase obj
    );

    Session create_basic_session(
        in AccessMode access_mode,
        in TypeId catalog_type_name,
        in ParameterList additional_parameters
    );

    TransactionalSession create_transactional_session(
        in AccessMode access_mode,
        in IsolationLevel default_isolation_level,
        in EndOfAssociationCallback callback,
        in TypeId catalog_type_name,
        in ParameterList additional_parameters
    );

    TransactionalSession current_session();

    TransactionalSessionList sessions(
```

```
        in CosTransactions::Coordinator transaction
    );

    StorageObjectFactory register_storage_object_factory(
        in TypeId storage_type_name,
        in StorageObjectFactory storage_object_factory
    );

    StorageHomeFactory register_storage_home_factory(
        in TypeId storage_home_type_name,
        in StorageHomeFactory storage_home_factory
    );

    SessionFactory register_session_factory(
        in TypeId catalog_type_name,
        in SessionFactory session_factory
    );

    SessionPoolFactory register_session_pool_factory(
        in TypeId catalog_type_name,
        in SessionPoolFactory session_pool_factory
    );

};
```

Connector::create_basic_session()

```
// PSDL code
Session create_basic_session(
    in AccessMode access_mode,
    in TypeId catalog_type_name,
    in ParameterList additional_parameters
);
```

Creates a basic, non-transactional session and returns a reference to the session.

Parameters

`access_mode`	The access can be read-only or both read and write.
`catalog_type_name`	The value is either an empty string or the PSDL type id of a catalog. For example: `PSDL:com/acme/People:1.0`.
`additional_parameters`	See Table 2.

Table 2: *Additional PSS Session Creation Parameters*

Parameter Name	Type	Description
`to`	`string`	This is a required parameter. Some string that identifies what you connect to. For example with PSS/DB, it will be an environment name; with PSS/ODBC a datasource name; with PSS/Oracle, an Oracle database name.
`concurrent`	`boolean`	Will this session be used by multiple concurrent threads? This parameter is not required. The default value is false.
`single writer`	`boolean`	Is this session the only session that writes to this database? When true, there is no risk of deadlock and the cache can be kept as-is after a commit. This parameter is not required. The default value is false.
Additional Relational Parameters		
`pessimistic locking`	`boolean`	Does this session acquire a write lock before updating an object in its cache? The default value is true. This parameter is not required.
`incarnation map size`	`long`	The size of the per-session hash map in which PSS/R keeps incarnations. The given value is rounded up to the closest power of 2. The default value is `1024`. This parameter is not required.

Exceptions

PERSIST_STORE	A session cannot be provided with the desired (or higher) access mode.

See Also CosPersistentState::Connector::create_transactional_session()

Connector::create_transactional_session()

```
// PSDL code
TransactionalSession create_transactional_session(
    in AccessMode access_mode,
    in IsolationLevel default_isolation_level,
    in EndOfAssociationCallback callback,
    in TypeId catalog_type_name,
    in ParameterList additional_parameters
);
```

Creates a new transactional session and returns a reference to the session.

Parameters

access_mode	The access can be read-only or both read and write.
default_isolation_level	The isolation level of resources created by this transactional session.
callback	Your application can be notified when a session is released by PSS by passing in an EndOfAssociationCallback local object.
catalog_type_name	The value is either an empty string or the PSDL type id of a catalog. For example: PSDL:com/acme/People:1.0.
additional_parameters	See Table 3.

Table 3: *Additional PSS TransactionalSession Creation Parameters*

Parameter Name	Type	Description
`to`	`string`	This is a required parameter. Some string that identifies what you connect to. For example with PSS/DB, it will be an environment name; with PSS/ODBC a datasource name; with PSS/Oracle, an Oracle database name.
`concurrent`	`boolean`	Will this session be used by multiple concurrent threads? This parameter is not required. The default value is false.
`single writer`	`boolean`	Is this session the only session that writes to this database? When true, there is no risk of deadlock and the cache can be kept as-is after a commit. This parameter is not required. The default value is false.

Exceptions

`PERSIST_STORE` Raised if:

- The session cannot be provided with the desired (or higher) access mode.
- The implementation cannot provide the desired default isolation level.

See Also `CosPersistentState::Connector::create_basic_session()`

Connector::current_session()

```
// PSDL code
TransactionalSession current_session();
```

Returns the current transactional session. The operation logically calls `sessions()` with the transaction associated with the calling thread.

Exceptions

`PERSIST_STORE` A single session cannot be returned.

See Also CosPersistentState::Connector::sessions()

Connector::get_pid()

```
// PSDL code
Pid get_pid(
    in StorageObjectBase obj
);
```

Returns the Pid of the given storage object.

See Also CosPersistentState::Connector::get_short_pid()

Connector::get_short_pid()

```
// PSDL code
ShortPid get_short_pid(
    in StorageObjectBase obj
);
```

Returns the ShortPid of the given storage object.

See Also CosPersistentState::Connector::get_pid()

Connector::implementation_id Attribute

```
// PSDL code
readonly attribute string implementation_id;
```

Returns the Id of this implementation.

Connector::register_session_factory()

```
// PSDL code
SessionFactory register_session_factory(
     in TypeId catalog_type_name,
     in SessionFactory session_factory
);
```

Registers a session factory and returns the factory previously registered with the given name. The operation returns NULL when there is no previously registered factory.

See Also

CosPersistentState::Connector::register_storage_object_factory()
CosPersistentState::Connector::register_storage_home_factory()
CosPersistentState::Connector::register_session_pool_factory()

Connector::register_session_pool_factory()

```
// PSDL code
SessionPoolFactory register_session_pool_factory(
     in TypeId catalog_type_name,
     in SessionPoolFactory session_pool_factory
);
```

Registers session pool factories and returns the factory previously registered with the given name. The operation returns NULL when there is no previously registered factory.

See Also

CosPersistentState::Connector::register_storage_object_factory()
CosPersistentState::Connector::register_storage_home_factory()
CosPersistentState::Connector::register_session_factory()

Connector::register_storage_home_factory()

```
// PSDL code
StorageHomeFactory register_storage_home_factory(
     in TypeId storage_home_type_name,
     in StorageHomeFactory storage_home_factory
);
```

Registers storage home factories and returns the factory previously registered with the given name. The operation returns NULL when there is no previously registered factory.

See Also

CosPersistentState::Connector::register_storage_object_factory()
CosPersistentState::Connector::register_session_factory()
CosPersistentState::Connector::register_session_pool_factory()

Connector::register_storage_object_factory()

```
// PSDL code
StorageObjectFactory register_storage_object_factory(
     in TypeId storage_type_name,
     in StorageObjectFactory storage_object_factory
);
```

Registers storage object factories and returns the factory previously registered with the given name. The operation returns NULL when there is no previously registered factory.

See Also

CosPersistentState::Connector::register_storage_home_factory()
CosPersistentState::Connector::register_session_factory()
CosPersistentState::Connector::register_session_pool_factory()

Connector::sessions()

```
// PSDL code
TransactionalSessionList sessions(
    in CosTransactions::Coordinator transaction
);
```

Returns all the transactional sessions created by this connector that are associated with resources registered with the given transaction. Very often `sessions()` returns a single session.

See Also

CosPersistentState::Connector::current_session()

CosPersistentState_Factory Template

The `CosPersistentState_Factory` class is a helper template you use to build `StorageHomeFactory` and `StorageObjectFactory` objects. The class contains the following virtual methods.

```
template <class T>
class CosPersistentState_Factory {
  public:

    virtual T* create()
        throw(CORBA::SystemException) = 0;

    virtual void _add_ref() {}

    virtual void _remove_ref() {}

    virtual ~CosPersistentState_Factory() {}
};
```

CosPersistentState:: EndOfAssociationCallback Interface

The `EndOfAssociationCallback` interface is implemented by the developer of the application. When a session-resource association has ended, the session may not become available immediately. For example, if the session is implemented using an ODBC or JDBC connection, PSS needs this connection until the resource (ODBC/JDBC transaction) is committed or rolled back.

```
// PSDL code in module CosPersistentState
local interface EndOfAssociationCallback {
    void released(in TransactionalSession session);
};
```

See Also `CosPersistentState::Connector::create_transactional_session()`

CosPersistentState::Session Interface

A PSS session is a logical connection between a process and one or more datastores. There are two kinds of sessions:

- Basic sessions for file-like access.
- Transactional sessions for transactional access. (See the `TransactionalSession` interface.)

You create a basic session by calling `Connector::create_basic_session()`. A basic session is a local object that supports the following interface:

```
// PSDL Code in module CosPersistentState
local interface Session : CatalogBase {};
```

See Also `IT_PSS::Session`

CosPersistentState:: StorageHomeBase Interface

A storage home can have behavior that is described by operations on its abstract storage home(s). An abstract storage home can also define any number of keys; each key declaration implicitly declares a pair of finder operations. All storage home instances implement the local interface `StorageHomeBase`:

```
// PSDL in module CosPersistentState
local interface StorageHomeBase {

    StorageObjectBase find_by_short_pid(
        in ShortPid short_pid
    )
    raises (NotFound);

    CatalogBase get_catalog();
};
```

StorageHomeBase::find_by_short_pid()

```
// PSDL code
StorageObjectBase find_by_short_pid(
    in ShortPid short_pid
)
    raises (NotFound);
```

Returns a storage object for the given short pid.

Parameters

`short_pid`	The short pid in the target storage home.

Exceptions

CosPersistentState::NotFound	The object is not found.

StorageHomeBase::get_catalog()

```
// PSDL code
    CatalogBase get_catalog();
```

Returns the catalog that manages the target storage home instance.

CosPersistentState:: StorageHomeFactory Native Type

The `StorageHomeFactory` is a native PSDL type.

```
// PSDL in module CosPersistentState
native StorageHomeFactory;
```

The C++ mapping of this native type is as follows:

```
// C++
typedef CosPersistentState_Factory<StorageHomeBase>
   StorageHomeFactory;
```

The application developer derives a class from this `StorageHomeFactory` type to provide an implementation.

See Also `IT_PSS_StorageHomeFactory`
`CosPersistentState::CosPersistentState_Factory`

CosPersistentState::StorageObject Interface

The `StorageObject` interface supports a PSS storage object.

```
// PSDL in module CosPersistentState
abstract storagetype StorageObject {
    void destroy_object();
    boolean object_exists();
    Pid get_pid();
    ShortPid get_short_pid();
    StorageHomeBase get_storage_home();
};
```

StorageObject::destroy_object()

```
// PSDL code
void destroy_object();
```

When called on an incarnation, the operation destroys the associated storage object (but does not destroy any of its incarnation).

Exceptions

`PERSIST_STORE` The operation is called on the instance of an embedded storage object.

StorageObject::get_pid()

```
// PSDL code
Pid get_pid();
```

Returns the `Pid` of the associated storage object when called on an incarnation.

Exceptions

PERSIST_STORE	The operation is called on the instance of an embedded storage object.

See Also CosPersistentState::StorageObject::get_short_pid()

StorageObject::get_short_pid()

```
// PSDL code
ShortPid get_short_pid();
```

Returns the ShortPid of the associated storage object when called on an incarnation.

Exceptions

PERSIST_STORE	The operation is called on the instance of an embedded storage object.

See Also CosPersistentState::StorageObject::get_pid()

StorageObject::get_storage_home()

```
// PSDL code
StorageHomeBase get_storage_home();
```

Returns the storage home instance that manages the target storage object instance.

StorageObject::object_exists()

```
// PSDL code
boolean object_exists();
```

Returns true if the target incarnation represents an actual storage object and false if it does not.

CosPersistentState:: StorageObjectBase Native Type

A storage object can have both state and behavior. The visible part of its state is described by state members on its abstract storage type(s). Similarly, its behavior is described by operations on its abstract storage type(s).

All storage object instances are derived from this common base, `StorageObjectBase`:

```
// PSDL in module CosPersistentState
    native StorageObjectBase;
```

The C++ mapping of this native type is as follows:

```
class StorageObjectBase {
protected:
    virtual ~StorageObjectBase() {}
};
```

CosPersistentState:: StorageObjectFactory Native Type

`StorageObjectFactory` is a native type.

```
// in module CosPersistentState
native StorageObjectFactory;
```

The C++ mapping of this native type is as follows:

```
// C++
typedef CosPersistentState_Factory<StorageObject>
   StorageObjectFactory;
```

The application developer derives a class from this `StorageObjectFactory` type to provide an implementation.

See Also

`IT_PSS_StorageObjectFactory`
`CosPersistentState::CosPersistentState_Factory`

CosPersistentState::StorageObjectRef Class

The StorageObjectRef class is a standard C++ base class mapping for a StorageObject reference.

```
class StorageObjectRef {
  public:
    typedef StorageObject _target_type;

    static CORBA::TypeCode_ptr _static_type();

    StorageObjectRef(
        StorageObject*  obj       = 0,
        CatalogBase_ptr catalog   = 0,
        void*           impl_data = 0
    );
    StorageObjectRef(
        const StorageObjectRef& ref
    );

    StorageObjectRef&  operator=(
        const StorageObjectRef& ref
    );

    StorageObjectRef&  operator=(
        StorageObject* obj
    );

    void release();

    StorageObject* operator->(); // not const!

    CORBA::Boolean same_ref(
        StorageObjectRef
    ) const;

    void destroy_object() const;
```

```
    Pid* get_pid() const;

    ShortPid* get_short_pid() const;

    CORBA::Boolean is_null() const;

    StorageHomeBase_ptr get_storage_home() const;


    // read-only access to data members

    void* _impl_data() const;

    CosPersistentState::CatalogBase_ptr _catalog() const;

    StorageObject* _target() const;

  protected:
    CosPersistentState::CatalogBase_ptr m_catalog;
    void*                               m_impl_data;
    StorageObject*                      m_target;
};
```

StorageObjectRef::_catalog()

`CosPersistentState::CatalogBase_ptr _catalog() const;`

Returns the catalog of the object.

StorageObjectRef::destroy_object()

`void destroy_object() const;`

Destroys the target object.

StorageObjectRef::get_pid()

```
Pid* get_pid() const;
```

Returns the Pid of the target object.

StorageObjectRef::get_short_pid()

```
ShortPid* get_short_pid() const;
```

Returns the short pid of the target object.

StorageObjectRef::get_storage_home()

```
StorageHomeBase_ptr get_storage_home() const;
```

Returns the storage home of the target object.

StorageObjectRef::_impl_data()

```
void* _impl_data() const;
```

StorageObjectRef::is_null()

```
CORBA::Boolean is_null() const;
```

Returns true if and only if this reference is null.

StorageObjectRef::operator=()

```
StorageObjectRef&  operator=(
    const StorageObjectRef& ref
);
```

An assignment operator that takes an incarnation of the target abstract storage type.

```
StorageObjectRef&  operator=(
    StorageObject* obj
);
```

An assignment operator.

StorageObjectRef::operator->()

```
StorageObject* operator->(); // not const!
```

A de-reference operator that de-references this reference and returns the target object. The caller is not supposed to release this incarnation.

StorageObjectRef::release()

```
void release();
```

Releases the reference.

StorageObjectRef::same_ref()

```
CORBA::Boolean same_ref(
    StorageObjectRef
) const;
```

Returns true if the input storage object reference is the same as this one.

StorageObjectRef::_static_type()

```
static CORBA::TypeCode_ptr _static_type();
```

Returns a TypeCode reference.

StorageObjectRef::StorageObjectRef() Constructors

```
StorageObjectRef(
    StorageObject*  obj       = 0,
    CatalogBase_ptr catalog   = 0,
```

```
    void*           impl_data = 0
);
```

The default constructor creates a null reference.

```
StorageObjectRef(
    const StorageObjectRef& ref
);
```

A non-explicit constructor that takes an incarnation of the target abstract storage type.

StorageObjectRef::_target_type

```
typedef StorageObject _target_type;
```

A type definition to the target type. This is useful for programming with templates.

StorageObjectRef::_target()

```
StorageObject* _target() const;
```

Returns the target object.

CosPersistentState:: TransactionalSession Interface

A transactional session is a specialized session that provides transactional access to storage objects. A transactional session supports the local interface `TransactionalSession`.

At a given time, a transactional session can be associated with one resource object (a datastore transaction), or with no resource at all. The session-resource association can be active, suspended, or ending. The state members of an incarnation managed by a transactional session can be used only when this session has an active association with a resource.

Typically, a resource is associated with a single session for its entire lifetime. However, with some advanced database products, the same resource may be associated with several sessions, possibly at the same time.

You create a transaction session by calling `create_transactional_session()`.

```
// PSDL Code in module CosPersistentState
local interface TransactionalSession : Session {

    readonly attribute IsolationLevel default_isolation_level;

    typedef short AssociationStatus;
    const AssociationStatus NO_ASSOCIATION = 0;
    const AssociationStatus ACTIVE         = 1;
    const AssociationStatus SUSPENDED      = 2;
    const AssociationStatus ENDING         = 3;

    void start(in CosTransactions::Coordinator transaction);
    void suspend(in CosTransactions::Coordinator transaction);
    void end(
        in CosTransactions::Coordinator transaction,
        in boolean success
    );
```

```
    AssociationStatus get_association_status();

    CosTransactions::Coordinator transaction();
    };
```

See Also IT_PSS::TransactionalSession
CosPersistentState::Session

TransactionalSession::AssociationStatus Type

```
// PSDL Code
typedef short AssociationStatus;
const AssociationStatus NO_ASSOCIATION = 0;
const AssociationStatus ACTIVE         = 1;
const AssociationStatus SUSPENDED      = 2;
const AssociationStatus ENDING         = 3;
```

The association status of a resource with a session. Valid values include:

NO_ASSOCIATION
ACTIVE
SUSPENDED
ENDING

See Also CosPersistentState::TransactionalSession::
get_association_status()

TransactionalSession::default_isolation_level Attribute

```
// PSDL Code
readonly attribute IsolationLevel default_isolation_level;
```

Returns the default isolation level of resources created for this transactional session.

TransactionalSession::end()

```
// PSDL Code
void end(
    in CosTransactions::Coordinator transaction,
    in boolean success
```

```
);
```

Terminates a session-transaction association.

Parameters

`transaction`	The transaction of the resource.
`success`	If the success parameter is FALSE, the resource is rolled back immediately. Like `refresh()`, `end()` invalidates direct references to incarnations' data members.

A resource can be prepared or committed in one phase only when it is not actively associated with any session. The resource will rollback if it is asked to prepare or commit in one phase when still in use. A resource ends any session-resource association in which it is involved when it is prepared, committed in one phase, or rolled back.

Exceptions

`PERSIST_STORE`	No associated resource.
`INVALID_TRANSACTION`	The given transaction does not match the transaction of the resource associated with this session.

The standard exception is raised if

See Also `CosPersistentState::TransactionalSession::start()`
`CosPersistentState::TransactionalSession::suspend()`

TransactionalSession::get_association_status()

```
// PSDL Code
AssociationStatus get_association_status();
```

Returns the status of the association (if any) with this session.

TransactionalSession::start()

```
// PSDL Code
void start(
    in CosTransactions::Coordinator transaction
);
```

Starts the transaction.

Parameters

`transaction` The transaction to start.

This operation does one of three things depending on the association of the transaction:

1. When `transaction` matches the transaction of the suspended (or ending) association, `start()` re-activates a suspended (or ending) session-resource association.
2. If a resource compatible with this session is already associated with the given transaction, `start()` associates this resource with this session, and makes the association active.
3. If the session creates a new resource and registers it with the given transaction. The session also associates itself with this resource and makes the association active.

Exceptions

`INVALID_TRANSACTION` There is a suspended (or ending) association but the transactions do not match.

See Also `CosPersistentState::TransactionalSession::suspend()`
`CosPersistentState::TransactionalSession::end()`

TransactionalSession::suspend()

```
// PSDL Code
void suspend(
    in CosTransactions::Coordinator transaction
);
```

Suspends a session-resource association.

Parameters

`transaction` The transaction to suspend.

Exceptions

`PERSIST_STORE` No active association.

The standard exception `INVALID_TRANSACTION` is raised if the given transaction does not match the transaction of the resource actively associated with this session.

See Also `CosPersistentState::TransactionalSession::start()`
`CosPersistentState::TransactionalSession::end()`

TransactionalSession::transaction()

```
// PSDL Code
CosTransactions::Coordinator transaction();
```

Returns the coordinator of the transaction with which the resource associated with this session is registered. The operation returns a nil object reference when the session is not associated with a resource.

IT_PSS Overview

The IT_PSS interfaces consist of:

CatalogBase
Connector
PreparedStatement
Master
Replica
ResultSet
Session
SessionManager
Statement
StorageObject
TransactionalSession

This module also has the following helper classes:

IT_PSS_StorageHomeFactory
IT_PSS_StorageObjectFactory
TxSessionAssociation

IT_PSS::TxSessionAssociation Class

You can use stack-allocated `TxSessionAssociation` objects to create associations between OTS transactions and PSS transactional sessions managed by a `SessionManager`.

```
class  TxSessionAssociation {
  public:
    TxSessionAssociation(
        IT_PSS::SessionManager_ptr              session_mgr,
        CosPersistentState::AccessMode          access_mode
    ) throw(CORBA::SystemException);

    TxSessionAssociation(
        IT_PSS::SessionManager_ptr              session_mgr,
        CosPersistentState::AccessMode          access_mode,
        CosTransactions::Coordinator_ptr        tx_coordinator
    ) throw(CORBA::SystemException);

    ~TxSessionAssociation()
        throw(CORBA::SystemException);

    IT_PSS::TransactionalSession_ptr  get_session_nc()
        const throw();

    CosTransactions::Coordinator_ptr  get_tx_coordinator_nc()
        const throw();

    void suspend()
        throw(CORBA::SystemException);

    void end(
        CORBA::Boolean success = IT_TRUE
    ) throw(CORBA::SystemException);

  private:
    ...
};
```

Enhancement This class is an Orbix enhancement.

TxSessionAssociation::end()

```
void end(
    CORBA::Boolean success = IT_TRUE
) throw(CORBA::SystemException);
```

Ends the association only if this object started or resumed the association.This method has no effect if the association already ended.

Parameters

success	Determines if the method was successful.

Enhancement This is an Orbix enhancement.

See Also `IT_PSS::TxSessionAssociation::suspend()`

TxSessionAssociation::get_session_nc()

```
IT_PSS::TransactionalSession_ptr  get_session_nc()
    const IT_THROW_DECL(());
```

Returns a non-copied reference to the session. This mean that the caller must not release the returned reference.

Enhancement This is an Orbix enhancement.

TxSessionAssociation::get_tx_coordinator_nc()

```
CosTransactions::Coordinator_ptr  get_tx_coordinator_nc()
    const IT_THROW_DECL(());
```

Returns a non-copied reference to the association's transaction coordinator. This mean that the caller must not release the returned reference. After a transaction-session association object is constructed, `get_tx_coordinator_nc()` returns nil when and only when the object represents an association between the session manager's read-only transaction and the session manager's shared read-only session.

Enhancement This is an Orbix enhancement.

TxSessionAssociation::TxSessionAssociation() Constructors

```
TxSessionAssociation(
    IT_PSS::SessionManager_ptr          session_mgr,
    CosPersistentState::AccessMode      access_mode
) throw(CORBA::SystemException);
```

A constructor without a supplied transaction.

```
TxSessionAssociation(
    IT_PSS::SessionManager_ptr          session_mgr,
    CosPersistentState::AccessMode      access_mode,
    CosTransactions::Coordinator_ptr    tx_coordinator
) throw(CORBA::SystemException);
```

A constructor with a transaction.

Parameters

`session_mgr`	The session manager.
`access_mode`	Access mode for the association. If `tx_coordinator` is not provided, the constructor's behavior is as follows: • If access mode is `READ_ONLY`, then start or use an association between the session manager's read-only transaction and the session manager's shared read-only session. • If access mode is `READ_WRITE`, then raise the `CORBA::TRANSACTION_REQUIRED`.
`tx_coordinator`	A transaction coordinator.

If a transaction is provided, the behavior depends on the number of associations between this transaction and sessions created by the session's manager connector:

Table 4: *Associations Between a Transaction and Sessions*

Number of Associations	Behavior
Greater than 1	Raises the `CORBA::IMPL_LIMIT` exception.
1	Does nothing if it is `ACTIVE`, otherwise it starts it.
none	Creates a new association between this transaction and a read-write transactional session managed by the session manager.

Enhancement This is an Orbix enhancement.

TxSessionAssociation::~TxSessionAssociation() Destructor

```
~TxSessionAssociation()
    throw(CORBA::SystemException);
```

If there is still an association when the destructor is called, and this object started the association, the association is suspended. If the suspend fails, the association ends with the success flag set to FALSE.

Enhancement This is an Orbix enhancement.

TxSessionAssociation::suspend()

```
void suspend()
    throw(CORBA::SystemException);
```

Suspends the association only when this object started or resumed the association. This method has no effect if the association has already suspended or ended.

Enhancement This is an Orbix enhancement.

See Also `IT_PSS::TxSessionAssociation::end()`

IT_PSS::TransactionalSession Interface

When you create a transactional session with an IONA PSS implementation, you get an `IT_PSS::TransactionalSession` object.

```
// PSDL Code in module IT_PSS
local interface TransactionalSession :
Session, CosPersistentState::TransactionalSession
{
Master get_master();
boolean is_replica();
Replica get_replica();
};
```

This interface provides propriatary enhancements to the OGM `TransactionalSession` interface. It consists of functions to manage replicated persistent objects.

IT_PSS::TransactionalSession::get_master

```
Master get_master();
```

Returns an object reference to a replica's master instance. If the session is associated with a master, then it will return an object reference to itself. If the master instance was not set or is unreachable, the function will return `NIL`.

IT_PSS::TransactionalSession:is_replica

```
boolean is_replica();
```

Returns `TRUE` if the object is a replica of a datastore and `FALSE` if it is not.

IT_PSS::TransactionalSession:get_replica

```
Replica get_replica();
```

If the session is associated with a replica of a datastore, it will return an object reference to its `Replica` object. If the session is associated with a master instance, it will return `NIL`.

IT_PSS::StorageObject Interface

PSS presents persistent information as storage objects. Each storage object has a type that defines its members and operations. When you create a storage object with an IONA PSS implementation, you get an `IT_PSS::StorageObject`.

```
// PSDL Code in module IT_PSS
abstract storagetype StorageObject {
    void it_lock();

    };
```

Enhancement This interface is an Orbix enhancement.

See Also `CosPersistentState::StorageObject`

StorageObject::it_lock()

```
// PSDL Code
void it_lock();
```

This operation acquires an exclusive lock on behalf of a basic session or transactional session.

Enhancement This is an Orbix enhancement.

IT_PSS::Statement Interface

The `Statement` interface provides operations for a JDBC-like statement, an object used for executing a static SQL statement and obtaining the results produced by it.

```
// PSDL Code in module IT_PSS
local interface Statement {
    void execute(
        in string pssql
    );

    ResultSet execute_query(
        in string pssql
    );

    unsigned long execute_update(
        in string pssql
    );

    ResultSet get_result_set();

    void close();

    // Default fetch direction and size
    //
    void set_fetch_direction(
        in ResultSet::FetchDirection direction
    );

    ResultSet::FetchDirection get_fetch_direction();

    void set_fetch_size(
        in unsigned short fetch_size
    );

    unsigned short get_fetch_size();

    // Type and Concurrency
```

```
    //
    ResultSet::Type get_result_set_type();
    ResultSet::Concurrency get_result_set_concurrency();
    CatalogBase get_catalog();
};
```

Enhancement This interface is an Orbix enhancement.

See Also IT_PSS::CatalogBase
IT_PSS::PreparedStatement

Statement::close()

```
// PSDL Code
void close();
```

Releases this Statement object's database and resources immediately instead of waiting for this to happen when it is automatically closed.

Enhancement This is an Orbix enhancement.

Statement::execute()

```
// PSDL Code
void execute(
    in string pssql
);
```

Executes an SQL statement that may obtain multiple results.

Enhancement This is an Orbix enhancement.

Statement::execute_query()

```
// PSDL Code
ResultSet execute_query(
    in string pssql
);
```

Executes an SQL statement that returns a single ResultSet.

Enhancement This is an Orbix enhancement.

Statement::execute_update()

```
// PSDL Code
unsigned long execute_update(
    in string pssql
);
```

Executes an SQL `INSERT`, `UPDATE` or `DELETE` statement.

Enhancement This is an Orbix enhancement.

Statement::get_catalog()

```
// PSDL Code
CatalogBase get_catalog();
```

Returns the catalog for this Statement.

Enhancement This is an Orbix enhancement.

Statement::get_fetch_direction()

```
// PSDL Code
ResultSet::FetchDirection get_fetch_direction();
```

Returns the direction for fetching rows from database tables that is the default for result sets generated from this `Statement` object.

Enhancement This is an Orbix enhancement.

Statement::get_fetch_size()

```
// PSDL Code
unsigned short get_fetch_size();
```

Returns the number of result set rows that is the default fetch size for result sets generated from this `Statement` object.

Enhancement This is an Orbix enhancement.

Statement::get_result_set()

```
// PSDL Code
ResultSet get_result_set();
```

Returns the current result as a `ResultSet` object.

Enhancement This is an Orbix enhancement.

Statement::get_result_set_concurrency()

```
// PSDL Code
ResultSet::Concurrency get_result_set_concurrency();
```

Returns the result set concurrency.

Enhancement This is an Orbix enhancement.

Statement::get_result_set_type()

```
// PSDL Code
ResultSet::Type get_result_set_type();
```

Returns the type of the `ResultSet`.

Enhancement This is an Orbix enhancement.

Statement::set_fetch_direction()

```
// PSDL Code
void set_fetch_direction(
    in ResultSet::FetchDirection direction
);
```

Sets a hint as to the direction in which the rows in a result set should be processed.

Enhancement This is an Orbix enhancement.

Statement::set_fetch_size()

```
// PSDL Code
void set_fetch_size(
    in unsigned short fetch_size
);
```

Gives a hint as to the number of rows that should be fetched from the database when more rows are needed.

Enhancement This is an Orbix enhancement.

IT_PSS::SessionManager Interface

PSS fully support transactions, and works with any compliant transaction service implementation. Unless you are developing a trivial demonstration program, you should use transactions when developing applications with PSS.

You can use a `SessionManager` object to manage transactional sessions. A common pattern when developing a transactional server using PSS is to use a shared read-only read-committed transactional session for simple read-only non-transactional requests. Of course, you can also create and manage your transactional sessions directly with the standard lower level PSS APIs from the `CosPersistentState` module.

```
//PSDL in module IT_PSS
local interface SessionManager {
    TransactionalSession get_shared_read_only_session_nc();
    void block_readers_until_idle();
};
```

Enhancement This interface is an Orbix enhancement.

See Also `IT_PSS::Connector::it_create_session_manager()`

SessionManager::get_shared_read_only_session_nc()

```
//PSDL code
TransactionalSession get_shared_read_only_session_nc();
```

Returns a shared, read-only transactional session. In this context, shared means the transactional session is usable by multiple threads.

Enhancement This is an Orbix enhancement.

SessionManager::block_readers_until_idle()

```
//PSDL code
void block_readers_until_idle();
```

Blocks new threads from using the shared, read-only transactional session until no thread is using the session.

Enhancement This is an Orbix enhancement.

IT_PSS::Session Interface

When you create a session with an IONA PSS implementation, you get an `IT_PSS::Session`.

```
// PSDL Code in module IT_PSS
local interface Session : CatalogBase, CosPersistentState::Session
    {};
```

Enhancement This interface is an Orbix enhancement.

See Also `IT_PSS::CatalogBase`
`CosPersistentState::Session`

IT_PSS::ResultSet Interface

The ResultSet interface provides access to a table of data similar to a JDBC result set. A ResultSet object is usually generated by executing a Statement or a PreparedStatement. A ResultSet maintains a cursor pointing to its current row of data. Initially the cursor is positioned before the first row.

Data types include:

Concurrency Type
FetchDirection Type
Type

Operations include:

absolute()	get_fetch_size()	next()
after_last()	get_row()	previous()
before_first()	get_statement()	refresh_row()
cancel_row_updates()	get_type()	relative()
close()	insert_row()	row_deleted()
delete_row()	is_after_last()	row_inserted()
find_state_member()	is_before_first()	row_updated()
first()	is_first()	set()
get()	is_last()	set_by_name()
get_by_name()	last()	set_fetch_direction()
get_concurrency()	move_to_current_row()	set_fetch_size()
get_fetch_direction()	move_to_insert_row()	update_row()

Enhancement This interface is an Orbix enhancement.

See Also IT_PSS::CatalogBase

```
// PSDL Code in module IT_PSS
local interface ResultSet {

    typedef unsigned short Type;
    const Type TYPE_FORWARD_ONLY =        1;
    const Type TYPE_SCROLL_INSENSITIVE =  2;
    const Type TYPE_SCROLL_SENSITIVE =    3;
```

```
typedef unsigned short Concurrency;
const Concurrency CONCUR_READ_ONLY = 1;
const Concurrency CONCUR_UPDATABLE = 2;

typedef unsigned short FetchDirection;
const FetchDirection FETCH_FORWARD = 1;
const FetchDirection FETCH_REVERSE = 2;
const FetchDirection FETCH_UNKNOWN = 3;

Statement get_statement();

// Basic operations
//
boolean next();
void close();

any get(
    in unsigned short index
);

any get_by_name(
    in string state_member_name
);

// Find state_member
//
unsigned short find_state_member(
   in string state_member_name
);

// Getting/setting the current row
//
boolean is_after_last();
boolean is_before_first();
boolean is_first();
boolean is_last();
void after_last();
void before_first();
boolean first();
boolean last();
unsigned short get_row();

boolean absolute(
```

```
    in short row
);

boolean relative(
    in short rows
);

boolean previous();
void move_to_insert_row();
void move_to_current_row();

// Fetch direction and size
//
void set_fetch_direction(
    in FetchDirection direction
);

FetchDirection get_fetch_direction();

void set_fetch_size(
    in unsigned short fetch_size
);

unsigned short get_fetch_size();

// Type and Concurrency
//
Type get_type();
Concurrency get_concurrency();

// Was row modified?
//
boolean row_updated();
boolean row_inserted();
boolean row_deleted();

// Write operations
//
void set(
    in unsigned short index,
    in any value
);
```

```
    void set_by_name(
        in string state_member_name,
        in any value
    );

    void insert_row();
    void update_row();
    void delete_row();
    void refresh_row();
    void cancel_row_updates();
};
```

ResultSet::absolute()

```
// PSDL Code
boolean absolute(
    in short row
);
```

Moves the cursor to the given row number in the result set.

Parameters

`row` If the row number is positive, the cursor moves to the given row number with respect to the beginning of the result set. The first row is row 1, the second is row 2, and so on.

If the given row number is negative, the cursor moves to an absolute row position with respect to the end of the result set. For example, calling absolute(-1) positions the cursor on the last row, absolute(-2) indicates the next-to-last row, and so on.

An attempt to position the cursor beyond the first/last row in the result set leaves the cursor before/after the first/last row, respectively.

Enhancement This is an Orbix enhancement.

ResultSet::after_last()

```
// PSDL Code
void after_last();
```

Moves the cursor to the end of the result set, just after the last row. Has no effect if the result set contains no rows.

Enhancement This is an Orbix enhancement.

ResultSet::before_first()

```
// PSDL Code
void before_first();
```

Moves the cursor to the front of the result set, just before the first row. Has no effect if the result set contains no rows.

Enhancement This is an Orbix enhancement.

ResultSet::cancel_row_updates()

```
// PSDL Code
void cancel_row_updates();
```

Cancels the updates made to a row in the table.

Enhancement This is an Orbix enhancement.

ResultSet::close()

```
// PSDL Code
void close();
```

Releases this `ResultSet` object's database and JDBC resources immediately instead of waiting for this to happen when it is automatically closed.

Enhancement This is an Orbix enhancement.

ResultSet::Concurrency Type

```
// PSDL Code
typedef unsigned short Concurrency;
const Concurrency CONCUR_READ_ONLY = 1;
const Concurrency CONCUR_UPDATABLE = 2;
```

The concurrency mode of the table. It can be read-only or updated.

Enhancement This is an Orbix enhancement.

ResultSet::delete_row()

```
// PSDL Code
void delete_row();
```

Deletes the current row from the table.

Enhancement This is an Orbix enhancement.

ResultSet::FetchDirection Type

```
// PSDL Code
typedef unsigned short FetchDirection;
const FetchDirection FETCH_FORWARD = 1;
const FetchDirection FETCH_REVERSE = 2;
const FetchDirection FETCH_UNKNOWN = 3;
```

Defines the direction of table row processing.

FETCH_FORWARD	The rows in a result set will be processed in a forward direction; first-to-last.
FETCH_REVERSE	The rows in a result set will be processed in a reverse direction; last-to-first.
FETCH_UNKNOWN	The order in which rows in a result set will be processed is unknown.

Enhancement This is an Orbix enhancement.

ResultSet::find_state_member()

```
// PSDL Code
unsigned short find_state_member(
   in string state_member_name
);
```

Returns the index for the given result set's state member name.

Enhancement This is an Orbix enhancement.

ResultSet::first()

```
// PSDL Code
boolean first();
```

Moves the cursor to the first row in the result set. Returns true if the cursor is on a valid row; false if there are no rows in the result set

Enhancement This is an Orbix enhancement.

ResultSet::get()

```
// PSDL Code
any get(
    in unsigned short index
);
```

Returns the value for the given parameter index.

Enhancement This is an Orbix enhancement.

See Also IT_PSS::ResultSet::set()

ResultSet::get_by_name()

```
// PSDL Code
any get_by_name(
    in string state_member_name
);
```

Returns the value for a state member given the member name.

Enhancement This is an Orbix enhancement.

See Also IT_PSS::ResultSet::set_by_name()

ResultSet::get_concurrency()

```
// PSDL Code
Concurrency get_concurrency();
```

Returns the concurrency value.

Enhancement This is an Orbix enhancement.

ResultSet::get_fetch_direction()

```
// PSDL Code
FetchDirection get_fetch_direction();
```

Returns the direction of table row processing.

Enhancement This is an Orbix enhancement.

See Also IT_PSS::ResultSet::set_fetch_direction()

ResultSet::get_fetch_size()

```
// PSDL Code
unsigned short get_fetch_size();
```

Returns the number of rows that are fetched from the database when more rows are needed for this result set.

Enhancement This is an Orbix enhancement.

See Also IT_PSS::ResultSet::set_fetch_size()

ResultSet::get_row()

```
// PSDL Code
unsigned short get_row();
```

Returns the current row number. The first row is number 1, the second number is 2, and so on.

Enhancement This is an Orbix enhancement.

ResultSet::get_statement()

```
// PSDL Code
Statement get_statement();
```

Returns the `Statement` that produced this `ResultSet` object.

Enhancement This is an Orbix enhancement.

ResultSet::get_type()

```
// PSDL Code
Type get_type();
```

Returns the type of this result set. The type is determined by the `Statement` that created the result set.

Enhancement This is an Orbix enhancement.

ResultSet::insert_row()

```
// PSDL Code
void insert_row();
```

Inserts a row.

Enhancement This is an Orbix enhancement.

ResultSet::is_after_last()

```
// PSDL Code
boolean is_after_last();
```

Returns true if the cursor is after the last row in the result set, false if it is not.

Enhancement This is an Orbix enhancement.

ResultSet::is_before_first()

```
// PSDL Code
boolean is_before_first();
```

Returns true if the cursor is before the first row in the result set, false if it is not.

Enhancement This is an Orbix enhancement.

ResultSet::is_first()

```
// PSDL Code
boolean is_first();
```

Returns true if the cursor is on the first row of the result set, false if it is not.

Enhancement This is an Orbix enhancement.

ResultSet::is_last()

```
// PSDL Code
boolean is_last();
```

Returns true if the cursor is on the last row of the result set, false if it is not.

Enhancement This is an Orbix enhancement.

ResultSet::last()

```
// PSDL Code
boolean last();
```

Moves the cursor to the last row in the result set and returns true if the cursor is on a valid row; false if there are no rows in the result set.

Enhancement This is an Orbix enhancement.

ResultSet::move_to_current_row()

```
// PSDL Code
void move_to_current_row();
```

Moves the cursor to the remembered cursor position, usually the current row. This operation has no effect if the cursor is not on the insert row.

Enhancement This is an Orbix enhancement.

ResultSet::move_to_insert_row()

```
// PSDL Code
void move_to_insert_row();
```

Moves the cursor to the insert row.

Enhancement This is an Orbix enhancement.

ResultSet::next()

```
// PSDL Code
boolean next();
```

Moves the cursor down one row from its current position. A ResultSet cursor is initially positioned before the first row; the first call to next makes the first row the current row; the second call makes the second row the current row, and so on.

Enhancement This is an Orbix enhancement.

ResultSet::previous()

```
// PSDL Code
boolean previous();
```

Moves the cursor to the previous row in the result set.

Enhancement This is an Orbix enhancement.

ResultSet::refresh_row()

```
// PSDL Code
void refresh_row();
```

Refreshes the current row with its most recent value in the database. This cannot be called when the cursor is on the insert row.

Enhancement This is an Orbix enhancement.

ResultSet::relative()

```
// PSDL Code
boolean relative(
    in short rows
);
```

Moves the cursor a relative number of rows, either positive or negative. Attempting to move beyond the first/last row in the result set positions the cursor before/after the first/last row. Calling `relative(0) is` valid, but does not change the cursor position.

Enhancement This is an Orbix enhancement.

ResultSet::row_deleted()

```
// PSDL Code
boolean row_deleted();
```

Indicates whether a row has been deleted. A deleted row may leave a visible "hole" in a result set. This operation can be used to detect holes in a result set. The value returned depends on whether or not the result set can detect deletions.

Enhancement This is an Orbix enhancement.

ResultSet::row_inserted()

```
// PSDL Code
boolean row_inserted();
```

Indicates whether the current row has had an insertion. The value returned depends on whether or not the result set can detect visible inserts. The operation returns true if a row has had an insertion and insertions are detected.

Enhancement This is an Orbix enhancement.

ResultSet::row_updated()

```
// PSDL Code
boolean row_updated();
```

Indicates whether the current row has been updated. The value returned depends on whether or not the result set can detect updates. If the set can detect updates, the operation returns true if the row has been visibly updated by the owner or another.

Enhancement This is an Orbix enhancement.

ResultSet::set()

```
// PSDL Code
void set(
    in unsigned short index,
    in any value
);
```

Sets the value and parameter index.

Enhancement This is an Orbix enhancement.

See Also IT_PSS::ResultSet::get()

ResultSet::set_by_name()

```
// PSDL Code
void set_by_name(
    in string state_member_name,
    in any value
);
```

Sets the value for an object's member given the name of the member.

Enhancement This is an Orbix enhancement.

See Also IT_PSS::ResultSet::get_by_name()

ResultSet::set_fetch_direction()

```
// PSDL Code
void set_fetch_direction(
    in FetchDirection direction
);
```

Sets a hint as to the direction in which the rows in this result set will be processed. The initial value is determined by the statement that produced the result set. The fetch direction may be changed at any time.

Enhancement This is an Orbix enhancement.

See Also IT_PSS::ResultSet::get_fetch_direction()

ResultSet::set_fetch_size()

```
// PSDL Code
void set_fetch_size(
    in unsigned short fetch_size
);
```

The fetch size is a hint as to the number of rows that should be fetched from the database when more rows are needed for this result set. The default value is set by the Statement that created the result set. The fetch size may be changed at any time.

Parameters

`fetch_size`	If the fetch size is zero, a best guess is used.

Enhancement This is an Orbix enhancement.

See Also IT_PSS::ResultSet::get_fetch_size()

ResultSet::Type

```
// PSDL Code
typedef unsigned short Type;
const Type TYPE_FORWARD_ONLY =        1;
const Type TYPE_SCROLL_INSENSITIVE =  2;
const Type TYPE_SCROLL_SENSITIVE =    3;
```

The type of this result set. The type is determined by the Statement that created the result set.

Enhancement This is an Orbix enhancement.

ResultSet::update_row()

```
// PSDL Code
void update_row();
```

Updates the underlying database with the new contents of the current row. Cannot be called when the cursor is on the insert row.

Enhancement This is an Orbix enhancement.

IT_PSS:Replica Interface

The Replica interface provides functionality for replicated databases. The persistent state service supports two styles of replicas: a push-style replica and a pull-style replcia. A push-style replica is updated by the master instance of the object. A pull-style replica requests updates periodically from the master instance of the object.

```
interface Replica
{
  boolean set_master(in Master new_master);

  readonly attribute unsigned long long last_successful_refresh;

  // Pull refresh now
  void refresh();
};
```

IT_PSS::Replica::set_master

```
boolean set_master(in Master new_master)
```

Registers the replica with a master instance of the object. It returns TRUE if the registration is successful.

Parameters This function takes an object of `Master` containing an object reference to the master instance of the object.

IT_PSS::Replica::last_successful_refresh

```
readonly attribute unsigned long long last_successful_refresh
```

Returns the amount of time that has passed since the last time the replica was successfully refreshed by the master instance of the object.

IT_PSS:Replica:refresh

```
void refresh()
```

Requests an update from the master instance of the object. The master will completely sync the replica as a result of this call.

IT_PSS::PreparedStatement Interface

The `PreparedStatement` interface is a JDBC-like prepared statement which is an object that represents a pre-compiled SQL statement. An SQL statement is pre-compiled and stored in the `PreparedStatement` object so your application can then efficiently execute the statement multiple times.

```
// PSDL Code in module IT_PSS
local interface PreparedStatement : Statement {

    void execute_prepared();

    ResultSet execute_prepared_query();

    unsigned long execute_prepared_update();

    void define_parameter(
        in unsigned short parameter_index,
        in any parameter_value
    );

    void clear_parameters();
};
```

Enhancement This is an Orbix enhancement.

See Also IT_PSS::CatalogBase
IT_PSS::Statement

PreparedStatement::clear_parameters()

```
// PSDL Code
void clear_parameters();
```

Clears the current parameter values immediately.

Enhancement This is an Orbix enhancement.

PreparedStatement::define_parameter()

```
// PSDL Code
void define_parameter(
    in unsigned short parameter_index,
    in any parameter_value
);
```

Defines an SQL parameter value for the designated parameter index.

Enhancement This is an Orbix enhancement.

PreparedStatement::execute_prepared()

```
// PSDL Code
void execute_prepared();
```

Executes the prepared SQL statement.

Enhancement This is an Orbix enhancement.

See Also IT_PSS::PreparedStatement::execute_prepared_query()
IT_PSS::PreparedStatement::execute_prepared_update()

PreparedStatement::execute_prepared_query()

```
// PSDL Code
ResultSet execute_prepared_query();
```

Executes the SQL query in this `PreparedStatement` object and returns the result set generated by the query.

Enhancement This is an Orbix enhancement.

See Also IT_PSS::PreparedStatement::execute_prepared()
IT_PSS::PreparedStatement::execute_prepared_update()

PreparedStatement::execute_prepared_update()

```
// PSDL Code
unsigned long execute_prepared_update();
```

Executes the SQL INSERT, UPDATE or DELETE statement in this `PreparedStatement` object.

Enhancement This is an Orbix enhancement.

See Also `IT_PSS::PreparedStatement::execute_prepared()`
`IT_PSS::PreparedStatement::execute_prepared_query()`

IT_PSS:Master Interface

The Master interface provides functionality for master instances of replicated persistent objects using the persistent state service.

```
interface Master
{};
```

IT_PSS_StorageObjectFactory Template

Use this template class to help implement your StorageObjectFactory.

```
// c++
template<class T>
class IT_PSS_StorageObjectFactory :
public CosPersistentState::StorageObjectFactory {
  public:

    IT_PSS_StorageObjectFactory();

    virtual void  _add_ref();

    virtual void  _remove_ref();

    virtual CosPersistentState::StorageObject* create()
        throw(CORBA::SystemException);

  private:
    ...
};
```

Enhancement This is an Orbix enhancement.

IT_PSS_StorageObjectFactory::_add_ref()

```
virtual void _add_ref();
```

Increases the reference count by one.

IT_PSS_StorageObjectFactory::create()

```
virtual CosPersistentState::StorageObject* create()
        throw(CORBA::SystemException);
```

Creates and returns a new `StorageObject` object.

Enhancement This is an Orbix enhancement.

IT_PSS_StorageObjectFactory:: IT_PSS_StorageObjectFactory()

```
IT_PSS_StorageObjectFactory();
```

The constructor.

Enhancement This is an Orbix enhancement.

IT_PSS_StorageObjectFactory::_remove_ref()

```
virtual void _remove_ref();
```

Decreases the reference count by one.

Enhancement This is an Orbix enhancement.

IT_PSS_StorageHomeFactory Template

Use this template class to help implement your StorageHomeFactory.

```
template<class T>
class IT_PSS_StorageHomeFactory :
public CosPersistentState::StorageHomeFactory {
  public:

    IT_PSS_StorageHomeFactory();

    virtual void _add_ref();

    virtual void _remove_ref();

    virtual CosPersistentState::StorageHomeBase_ptr create()
        throw(CORBA::SystemException);

  private:
    ...
};
```

Enhancement This is an Orbix enhancement.

IT_PSS_StorageHomeFactory::_add_ref()

```
virtual void _add_ref();
```

Increases the reference count by one.

Enhancement This is an Orbix enhancement.

IT_PSS_StorageHomeFactory::create()

```
virtual CosPersistentState::StorageHomeBase_ptr create()
        throw(CORBA::SystemException);
```

Creates and returns a new StorageHomeBase object.

IT_PSS_StorageHomeFactory::IT_PSS_StorageHomeFactory()

```
IT_PSS_StorageHomeFactory();
```

The constructor.

Enhancement This is an Orbix enhancement.

IT_PSS_StorageHomeFactory::_remove_ref()

```
virtual void _remove_ref();
```

Decreases the reference count by one.

Enhancement This is an Orbix enhancement.

IT_PSS::Connector Interface

This is an Orbix-enhancement interface that lets you create a session manager.

```
// PSDL Code in module IT_PSS
/* local */ interface Connector : CosPersistentState::Connector {
    SessionManager it_create_session_manager(
        in CosPersistentState::ParameterList parameters
    );
};
```

Enhancement This is an Orbix enhancement.

See Also CosPersistentState::Connector

Connector::it_create_session_manager()

```
// PSDL Code
SessionManager it_create_session_manager(
    in CosPersistentState::ParameterList parameters
);
```

Creates and returns a session manager.

Parameters

`parameters`	See Table 5 for details about possible parameters. Other parameters are passed in each session creation call. You cannot, however, pass a parameter named `concurrent` when creating a session manager. The session manager's read-only read-committed session is created with concurrent set to true, whereas the session manager's read-write serializable sessions are created with concurrent set to false.

Table 5: *Additional PSS SessionManager Creation Parameters*

Parameter Name	Type	Description
`to`	`string`	This parameter is required. Some string that identifies what you connect to. For example with PSS/DB, it will be an environment name; with PSS/ODBC a datasource name; with PSS/Oracle, an Oracle database name.
`rw pool size`	`long`	Initial size of the pool of read-write transactional sessions managed by the session manager. Must be between 1 and 1000. This parameter is not required. The default value is 1.
`grow pool`	`boolean`	Create a new session to process a new request when all the read-write transactional sessions are busy? If false, wait until a read-write transactional session becomes available. This parameter is not required. The default value is false.
`single writer`	`boolean`	Can be true only when rw pool size is 1, in which case the read-write transactional session will be created with the single writer parameter set to true. This parameter is not required. The default value is false.

Enhancement This is an Orbix enhancement.

See Also `IT_PSS::SessionManager`

IT_PSS::CatalogBase Interface

PSS provides simple JDBC-like queries. You use `CatalogBase` to create a `Statement` or `PreparedStatement`.The query language is a subset of SQL that currently only supports the following form of select query:

```
select ref(h) from home_type_id h
```

The PSDL code is as follows:

```
// PSDL Code in Module IT_PSS
local interface CatalogBase :
CosPersistentState::CatalogBase {

    Statement it_create_statement();

    Statement it_create_statement_with_type_and_concurrency(
        in ResultSet::Type type,
        in ResultSet::Concurrency concurrency
    );

    PreparedStatement it_prepare_statement(
        in string pssql
    );

    PreparedStatement
    it_prepare_statement_with_type_and_concurrency(
        in string pssql,
        in ResultSet::Type type,
        in ResultSet::Concurrency concurrency
    );

    void it_discard_flush_list();

    void it_discard_all(
        in boolean clear_non_id_refs
    );
};
```

Enhancement This is an Orbix enhancement.

See Also

CosPersistentState::CatalogBase
IT_PSS::PreparedStatement
IT_PSS::Statement
IT_PSS::ResultSet

CatalogBase::it_create_statement()

```
// PSDL Code
Statement it_create_statement();
```

Creates and returns a JDBC-like Statement.

Enhancement This is an Orbix enhancement.

CatalogBase:: it_create_statement_with_type_and_concurrency()

```
// PSDL Code
Statement it_create_statement_with_type_and_concurrency(
    in ResultSet::Type type,
    in ResultSet::Concurrency concurrency
);
```

Creates and returns a JDBC-like Statement with a specific ResultSet type. The concurrency setting can be either read-only or updateable. Only one ResultSet per Statement can be open at any point in time. All statement execute methods implicitly close a statement's current ResultSet if an open one exists.

Enhancement This is an Orbix enhancement.

CatalogBase::it_discard_all()

```
// PSDL Code
void it_discard_all(
    in boolean clear_non_id_refs
);
```

Discards all cached objects.

Parameters

`clear_non_id_refs`	If this parameter is set to true, any references that an object might have to another object are removed. This removes the possibility of circular references between objects.

Enhancement This is an Orbix enhancement.

CatalogBase::it_discard_flush_list()

```
// PSDL Code
void it_discard_flush_list();
```

Discards all modified objects in the catalog.

Enhancement This is an Orbix enhancement.

CatalogBase::it_prepare_statement()

```
// PSDL Code
PreparedStatement it_prepare_statement(
    in string pssql
);
```

Creates and returns a JDBC-like `PreparedStatement` with the given query.

Enhancement This is an Orbix enhancement.

CatalogBase:: it_prepare_statement_with_type_and_concurrency()

```
// PSDL Code
PreparedStatement  it_prepare_statement_with_type_and_concurrency(
    in string pssql,
    in ResultSet::Type type,
    in ResultSet::Concurrency concurrency
);
```

Creates and returns a JDBC-like `PreparedStatement` with a given query and specific `ResutSet` type. The concurrency setting can be either read-only or updateable.

Enhancement This is an Orbix enhancement.

The IT_PSS_DB Module Overview

This module contains the single interface Env.

IT_PSS_DB::Env Interface

```
// IDL
module IT_PSS_DB {
    interface Env {
        readonly attribute string name;

        void pre_backup();
        void post_backup();
        void checkpoint();
    };
};
```

Enhancement This interface is an Orbix enhancement.

Env::checkpoint()

```
// IDL
void checkpoint();
```

Enhancement This is an Orbix enhancement.

Env::name Attribute

```
// IDL
readonly attribute string name;
```

Enhancement This is an Orbix enhancement.

Env::post_backup()

```
// IDL
void post_backup();
```

Enhancement This is an Orbix enhancement.

Env::pre_backup()

```
// IDL
void pre_backup();
```

Enhancement This is an Orbix enhancement.

Index

L

M

N

O

P

R

S

T

U

Y