Open Service Interface Definition

Shared

Document Release: 2.0
OSID version 2.0

Summary

The Shared Open Service Interface Definition (OSID) supports implementing a Type abstract class, and defining primitive-type iterators.

This OSID is called shared because many of the Services are used widely across other OSIDs. In particular: Id, Type, and TypeIterator are used extensively. Shared is also one of the first OSIDs one might implement. Without one's own implementation or another's, there is little one can do.

Service Definition

An examination of an Open Service Interface Definition (OSID) usually begins with the Manager, but the Shared OSID is unique among OSIDs in that it has no Manager class. All objects in the Shared OSID are created in other OSID Managers. Before looking at the OSID in detail, there are several concepts to review:

Ids

The Shared OSID contains the Id class, though creating, storing, and retrieving unique identifiers (Ids) is managed in the Id OSID.

In many contexts, people, organizations, objects, and processes are designated both a descriptive name and an unambiguous identifier. For example, students have names and student ids; courses have titles and course numbers; books have titles and ISBNs, web pages have names and URLs, etc. The convention is that names do not have to be unique but Ids should be, if not globally, at least within some broad context.

Types

OSIDs provides many methods that accept or return Types.[1] Types serve to categorize or classify objects. Each Type consists of three Strings: an authority, a domain, and a keyword. There is a fourth String, a description, which can be filled in but is not used when comparing one Type to another. Types employ these Strings as a nomenclature shared either between applications or between Service implementations and applications that employ those implementations. The OSIDs do not specify the content of the Type's Strings; that is left to the communities that develop implementations and applications.

Types can be used in a variety of ways. One use is to maintain certain constraints or integrity checks. For example, the DigitalRepository service might know about a certain AssetType:

authority	mit.edu
domain	digital_repository
keyword	image_asset

The DigitalRepository implementation expects Assets of a particular Type to contain certain information. When an application creates an Asset of a particular Type, the service can apply integrity rules based on the Type. For this use, since the Service implementation must understand all supported Types, permitting an application to create new Types would not make sense; the implementation would not know how to interpret them.

Another use is where applications use Types about which the implementations know nothing. An example is the QualifierType in Authorizations. An implementation does not need to know what to do with QualifierTypes; the implementation simply manages them by offering the application a way to use, maintain, and persist the Types.

authority	mit.edu
domain	authorization_qualifier
keyword	course_number_PHY101

Iterators

OSIDs provide many methods that return data. An examination of these methods shows that they make use of Iterators. A OSID Iterator provides sequential access to data of a particular type, one element at a time.[2] An iterator provides two methods: **hasNext**() indicates if more data elements are available to be returned; **next**() returns the next data element.

Some architectures use arrays to return data. OSIDs do not do this for two reasons. First, the size or number of data being returned might be large and therefore ill suited for assembly into an array for return. The iterator approach allows an implementation to return each data element separately. Second, the data being returned may come from a remote source[3] and make take some time to be fetched.[4] Using an iterator means that we do not have to assemble all the data before returning any of it. A tenet of the OSID design philosophy is that an application depending on an OSID implementation should not know the processing details of the implementation and implementations should be substitutable. Iterators allow data to

[1] The Shared OSID provides an abstract class for Type. This is one of the few instances of an abstract class rather than an interface. The abstract class provides a standard implementation of **isEqual()**, the method that compares Types.

[2] Note that in Java™ iterators return an object rather than a specific type.

[3] See also the Discussion of Serialization.

[4] Most means of returning results are synchronous and locking. There is an assumption that the process delivering data will not block. OSIDs make no such assumption.

be returned safely and dependably without dependence on particular implementation details of that process.

Iterators return a sequence of data elements, one at a time. For most of the OSID methods, there is no guarantee of the order in which elements will be returned. Iterators, unlike arrays or other data structures, do not allow access to elements by index; you must access values in sequence. There is no way to go backwards through the sequence[5]. All elements returned are of a specific type[6]. The Shared OSID defines iterators for primitive types such as Strings.[7] Individual OSIDs define type-specific iterators for objects they return in a squence. For example, the Authorization OSID defines an AuthorizationIterator.

osid.shared.Id

Ids have two methods: one returns a String representation and one compares the Id to another. The **getIdString**() method[8] returns a String representation of an Id. This can be convenient for persisting the Id and, of course, for displaying it. There is no requirement that the String be of a particular length and different implementations will likely provide different length String representations. Similarly, there is no requirement that these Strings contain all letters, all numerals, be of mixed case, etc.

The **isEqual**() method[9] accepts an Id and returns true[10] if the supplied Id matches this Id; false otherwise. Any implementation of this method is likely to simply compare two Ids' String representations.

Since every Id is unique, it is reasonable to ask why we would ever need to test two Ids for equivalence. Ids are input arguments to many methods, for example objects to be deleted are specified by their Id. Any implementation of such a delete method needs to be able to compare the Id provided with the Ids of objects previously created.

Method Summary

String	getIdString()
boolean	isEqual(osid.shared.Id id)

osid.shared.Properties

Properties provide for read-only access to a set of key-value pairs. Both keys and values are Serializable. Through this mechanism, Agents can have data values associated with them. Standardized Types here as elsewhere will promote interoperability.

[5] One could place the values returned in a data structure, such as an array, that allows for access by index.

[6] For the user of a Java™ language binding implementation, this approach avoids the need to cast the Object returned by Java™ iterators. An implementation might use a Java™ iterator and perform the cast within the implementation prior to returning the specifically typed object.

[7] It is recommend that no implementation of an OSID should implement an iterator with the intent of having that implementation reused by other OSID implementations. Doing so, would bind two OSIDs' implementations together.

[8] In the Java™ language binding, the OSID does not use the java.lang.Object method toString() since this method cannot throw an exception and all OSID methods throw exceptions by design.

[9] Similarly, OSID does not use equals() since this method cannot throw and exception.

[10] This is not a compare method as there is no Service definition for sorting Ids.

Method Summary

ObjectIterator	getKeys()
Serializable	getProperty(Serializable key)
Type	getType()

osid.shared.Type

Type is an abstract class with an final methods for getting the 4 String values that comprise the Type as well as an isEqual method for comparing them. These methods are final to avoid inconsistencies. There are two constructors, one with and one without the description String. The description String is not used to compare Types.

Method Summary

Type	Type(String domain, String authority, String keyword)
Type	Type(String domain, String authority, String keyword, String description)
String	getAuthority()
String	getDescription()
String	getDomain()
String	getKeyword()
boolean	isEqual(Type type)

osid.shared.TypeIterator and other Primitive-type Iterators

Iterators return a set of elements of a specific type, one at a time. The purpose of all Iterators is to offer a way for OSID methods to return multiple values of a common type and not use an array. Returning an array may not be appropriate if the number of values returned is large or is fetched remotely. Iterators do not allow access to values by index, rather you must access values in sequence. Similarly, there is no way to go backwards through the sequence unless you place the values in a data structure, such as an array, that allows for access by index.

The TypeIterator provides two methods: The **hasNext**() method returns true if there are more Types available; false otherwise. The **next**() method returns the next Type in the sequence. The order of Types is not guaranteed.

Method Summary

boolean	hasNext()
Type	next()

There are other Iterators defined in OSID Shared for common primitive types and objects. Specifically, there are iterators for Object, String, byte char, int. There are also iterators for Agent, Group, and Properties. These have the same methods as all other iterators.

Note that there are no definitions for other common types such as short, long, float, double, and boolean because they are not referenced in any OSID.

osid.shared.SharedException

The OSIDs make use of Exceptions as a mechanism for responding to error or unusual conditions. All methods in the Shared OSID throw a **SharedException**. The Exception contains a message that is a String. The following message Strings are defined in SharedException:

- ALREADY_ADDED
- CIRCULAR_ADDITION
- CONFIGURATION_ERROR
- ITERATOR_HAS_NO_MORE_ELEMENTS
- NULL_ARGUMENT
- OPERATION_FAILED
- PERMISSION_DENIED
- UNIMPLEMENTED
- UNKNOWN_ID
- UNKNOWN_KEY
- UNKNOWN_TYPE

If an implementation uses these messages, consumers of the implementation can easily test and conditionally respond to the Exception. Note that other kinds of Exception constructors are not used as all do or can devolve to a String. All methods of all interfaces of all OSIDs throw a subclass of osid.OsidException. This requires the caller of any implementation method handle the Exception.

If a method performs an operation without incident, an object or primitive may be returned, but in most cases, methods do not return error codes or a success or failure boolean. For example, a method that deletes an object with a particular identifier, would throw an Exception if the identifier were unknown; the method would not return, for example, false.