Service Data Objects For Java Specification

Version 2.01, November 2005

Authors

John Beatty	BEA Systems Inc.
Henning Blohm	SAP AG
Christophe Boutard	XCalia
Stephen Brodsky	IBM Corporation
Michael Carey	BEA Systems, Inc.
Jean-Jacques Dubray	SAP AG
Raymond Ellersick	IBM Corporation
Michael Ho	Sybase, Inc.
Anish Karmarkar	Oracle Corporation
Dan Kearns	Siebel Systems, Inc.
Regis Le Brettevillois	Xcalia
Martin Nally	IBM Corporation
Radu Preotiuc-Pietro	BEA Systems, Inc.
Mike Rowley	BEA Systems, Inc.
Shaun Smith	Oracle Corporation
Helena Yan	Siebel Systems, Inc.

Copyright Notice

License

The Service Data Objects Specification is being provided by the copyright holders under the following license. By using and/or copying this work, you agree that you have read, understood and will comply with the following terms and conditions:

Permission to copy and display the Service Data Objects Specification and/or portions thereof, without modification, in any medium without fee or royalty is hereby granted, provided that you include the following on ALL copies of the Service Data Objects Specification, or portions thereof, that you make:

1. A link or URL to the Service Data Objects Specification at these locations:

- http://dev2dev.bea.com/technologies/commonj/index.jsp
- http://www.ibm.com/developerworks/library/specification/ws-sdo/
- http://oracle.com/technology/webservices/sca
- https://www.sdn.sap.com/
- http://www.xcalia/xdn/specs/sdo
- http:/www.sybase.com/sca

2. The full text of this copyright notice as shown in the Service Data Objects Specification.

IBM, BEA, Oracle, SAP, Siebel, Xcalia, Sybase (collectively, the “Authors”) agree to grant you a royalty-free license, under reasonable, non-discriminatory terms and conditions to patents that they deem necessary to implement the Service Data Objects Specification.

THE Service Data Objects SPECIFICATION IS PROVIDED "AS IS," AND THE AUTHORS MAKE NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED, REGARDING THIS SPECIFICATION AND THE IMPLEMENTATION OF ITS CONTENTS, INCLUDING, BUT NOT LIMITED TO, WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, NON-INFRINGEMENT OR TITLE. THE AUTHORS WILL NOT BE LIABLE FOR ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF OR RELATING TO ANY USE OR DISTRIBUTION OF THE SERVICE DATA OBJECTS SPECIFICATION.

The name and trademarks of the Authors may NOT be used in any manner, including advertising or publicity pertaining to the Service Data Objects Specification or its contents without specific,

written prior permission. Title to copyright in the Service Data Objects Specification will at all times remain with the Authors.

No other rights are granted by implication, estoppel or otherwise.

Status of this Document

This specification may change before final release and you are cautioned against relying on the content of this specification. The authors are currently soliciting your contributions and suggestions. Licenses are available for the purposes of feedback and (optionally) for implementation.

Table Of Contents

Introduction

Service Data Objects (SDO) is a data programming **architecture** and an **API**.

The main purpose of SDO is to simplify data programming, so that developers can focus on business logic instead of the underlying technology.

SDO simplifies data programming by:

- unifying data programming across data source types
- providing support for common application patterns
- enabling applications, tools and frameworks to more easily query, view, bind, update, and introspect data.

For a high-level overview of SDO, see the white paper titled "Next-Generation Data Programming: Service Data Objects" [3].

Key Concepts

The key concepts in the SDO architecture are the **Data Object**, the **data graph** and the **Data Access Services (DAS)**.

A Data Object holds a set of named properties, each of which contains either a simple data-type value or a reference to another Data Object. The Data Object API provides a dynamic data API for manipulating these properties.

The data graph provides an envelope for Data Objects, and is the normal unit of transport between components. Data graphs can track changes made to the graph of Data Objects. Changes include inserting Data Objects, deleting Data Objects and modifying Data Object property values.

Usually, data graphs are constructed from one of the following:

- Data sources such as XML files, Enterprise Java™ Beans (EJBs), XML databases and relational databases.
- Services such as Web services, Java Connector Architecture (JCA) Resource Adapters and Java Message Service (JMS) messages.

Components that can populate data graphs from data sources and commit changes to data graphs back to the data source are called Data Access Services (DAS). The DAS architecture and APIs are outside the scope of this specification.

Requirements

The scope of the SDO specification includes the following requirements:

1. **Dynamic Data API.** Data Objects often have typed interfaces. However, sometimes it is either impossible or undesirable to create interfaces to represent the Data Objects. One common reason for this is when the data being transferred is defined by the output of a query. Examples would be:
   - A relational query against a relational persistence store.
   - An EJBQL queries against an EJB entity bean domain model.
   - Web services.
   - XML queries against an XML source.
   - When deployment of generated code is not practical.

   In these situations, it is necessary to use a dynamic store and associated API. SDO has the ability to represent Data Objects through a standard dynamic data API.

2. **Support for Static Data API.** In cases where metadata is known at development time (for example, the XML Schema definition or the SQL relational schema is known), SDO supports code-generating interfaces for Data Objects. When static data APIs are used, the dynamic data APIs are still available. SDO enables static data API code generation from a variety of metamodels, including:
   - Popular XML schema languages.
   - Relational database schemas with queries known at the time of code generation.
   - Web services, when the message is specified by an XML schema.
   - JCA connectors.
   - JMS message formats.
   - UML models

   While code-generation rules for static data APIs is outside the scope of this core SDO specification, it is the intent that SDO supports code-generated approaches for Data Objects.

3. **Complex Data Objects.** It is common to have to deal with "complex" or "compound" Data Objects. This is the case where the Data Object is the root of a tree, or even a graph of objects. An example of a tree would be a Data Object for an Order that has references to other Data Objects for the Line Items. If each of the Line Items had a reference to a Data Object for Product Descriptions, the set of objects would form a graph. When dealing with compound data objects, the change history is significantly harder to implement because inserts, deletes, adds, removes and re-orderings have to be tracked, as well as simple changes. Service Data Objects support arbitrary graphs of Data Objects with full change summaries.

4. **Change Summary.** It is a common pattern for a client to receive a Data Object from another program component, make updates to the Data Object, and then pass the modified Data Object back to the other program component. To support this scenario, it is often

important for the program component receiving the modified Data Object to know what modifications were made. In simple cases, knowing whether or not the Data Object was modified can be enough. For other cases, it can be necessary (or at least desirable) to know which properties were modified. Some standard optimistic collision detection algorithms require knowledge not only of which columns changed, but what the previous values were. Service Data Objects support full change summary.

5. **Navigation through graphs of data.** SDO provides navigation capabilities on the dynamic data API. All Data Objects are reachable by breadth-first or depth-first traversals, or by using a subset of XPath 1.0 expressions.

6. **Metadata.** Many applications are coded with built-in knowledge of the shape of the data being returned. These applications know which methods to call or fields to access on the Data Objects they use. However, in order to enable development of generic or framework code that works with Data Objects, it is important to be able to introspect on Data Object metadata, which exposes the data model for the Data Objects. SDO provides APIs for metadata. SDO metadata may be derived from:
   - XML Schema
   - EMOF (Essential Meta Object Facility)
   - Java
   - Relational databases
   - Other structured representations.

7. **Validation and Constraints.**
   - Supports validation of the standard set of constraints captured in the metadata. The metadata captures common constraints expressible in XML Schema and relational models (for example, occurrence constraints).
   - Provides an extensibility mechanism for adding custom constraints and validation.

8. **Relationship integrity.**
   - An important special case of constraints is the ability to define relationships between objects and to enforce the integrity of those constraints, including cardinality, ownership semantics and inverses. For example, consider the case where an employee has a relationship to its department and a department inversely has a list of its employees. If an employee's department identifier is changed then the employee should be removed, automatically, from the original department's list. Also, the employee should be added to the list of employees for the new department. Data Object relationships use regular Java objects as opposed to primary and foreign keys with external relationships.
   - Support for containment tree integrity is also important.

NOTE the following areas are out of scope:

9. **Complete metamodel and metadata API.** SDO includes a minimal metadata access API for use by Data Object client programmers. The intention is to provide a very simple client view of the model. For more complete metadata access, SDO may be used in conjunction with common metamodels and schema languages, such as XML Schema [1] and the EMOF compliance point from the MOF2 specification [2]. Java annotations in JSR 175 may be a future source of metadata.

10. **Data Access Service (DAS) specification.** Service Data Objects can be used in conjunction with "data accessors". Data accessors can populate data graphs with Data Objects from back-end data sources, and then apply changes to a data graph back to a data source. A data access service framework is out of scope but will be included in a future Data Access Service specification .

Organization of this Document

This specification is organized as follows:

- **Architecture**: Describes the overall SDO system.
- **API**: Defines and describes the Programming API for SDO.
- **Generating Code from XML Schemas**: Shows how code is generated from XML Schemas (XSD).
- **Interface Specification**: Defines how interfaces are generated and used.
- **Serialization of DataObjects**: Defines how to serialize DataObjects.
- **SDO Model for Types and Properties**: Shows the SDO Type and Property in model form.
- **Standard SDO Types**: Defines and describes the Standard SDO Types.
- **XML Schema to SDO Mapping**: Defines and describes how XML Schema declarations (XSD) are mapped to SDO Types and Properties.
- **Generation of XSD from SDO Type and Property**: Describes how to generate XSDs from SDO Types and Properties.
- **XPath Expression for DataObjects**: Defines an augmented subset of XPath that can be used with SDO for traversing through Data Objects.
- **Examples**: Provides a set of examples showing how SDO is used.
- **DataType Conversion Tables**: Shows the set of defined datatype conversions.

Architecture

The core of the SDO framework is the DataObject, which is a generic representation of a business object and is not tied to any specific persistent storage mechanism.

A data graph is used to collect a graph of related DataObjects. In SDO version 1.0 a data graph was always wrapped by a DataGraph envelope object, whereas in SDO version 2.0 a graph of DataObjects can exist outside of a DataGraph. When *data graph* is used as two lower case words, it refers to any set of DataObjects. When *DataGraph* is used as a single upper case word, it refers specifically to the DataGraph envelope object.

All data graphs have a single root DataObject that directly or indirectly contains all the other DataObjects in the graph. When all DataObjects in the data graph refer only to DataObjects in the data graph, then the data graph is called *closed*. *Closure* is the normal state for data graphs.

A data graph consists of:

- A single root DataObject.
- All the DataObjects that can be reached by recursively traversing the containment Properties of the root DataObject.

A closed data graph forms a tree of DataObjects, where the non-containment references point to DataObjects within the tree.

A data graph keeps track of the schema that describes the DataObjects. A data graph can also maintain a ChangeSummary, which represents the changes made to the DataObjects in the graph.

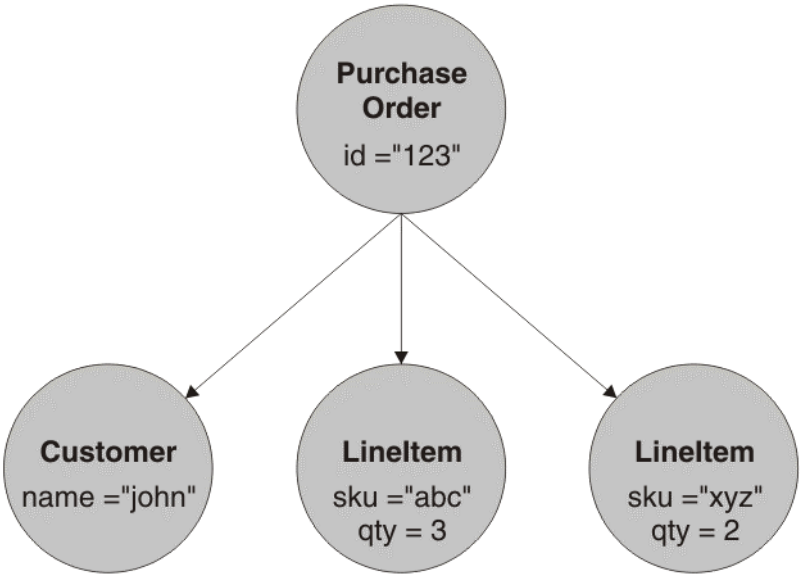


Figure 1: Data graph containing Data Objects

The standard way for an end user to get access to a data graph is through a Data Access Service (DAS). A DAS is a facility that provides methods to load a data graph from a store and to save a data graph back into that store. For example, an XML File DAS would load and save a data graph as an XML file and a JDBC DAS would load and save a data graph using a relational database. Specifications for particular DAS are outside the scope of this specification.

DAS typically uses a disconnected data architecture, whereby the client remains disconnected from the DAS except when reading a data graph or writing back a data graph. Thus, a typical scenario for using a data graph involves the following steps:

1. The end user sends a request to a DAS to load a data graph.
2. The DAS starts a transaction against the persistent store to retrieve data, creates a data graph that represents the data, and ends the transaction.
3. The DAS returns the data graph to an end user application.
4. The end user application processes the data graph.
5. The end user application calls the DAS with the modified data graph.
6. The DAS starts a new transaction to update the data in the persistent store based on the changes that were made by the end user.

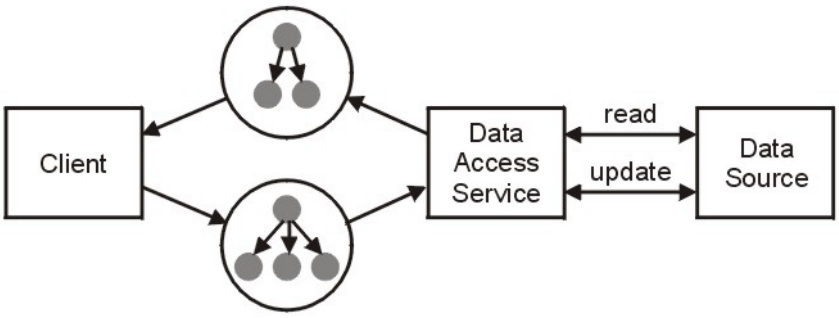


Figure 2: SDO's disconnected data architecture

Note that there are two distinct roles that can be identified among DataObject users: the client and the DAS writer.

The client needs to be able to traverse a data graph to access each DataObject and to get and set the fields in each DataObject. The client may also need to serialize and deserialize a data graph. Data graphs can be serialized to XML, typically by an XML DAS.

The DAS writer must be able to define a model for a data graph, create a new data graph, generate change history information, and access change history information. This specification’s focus is the perspective of the client.

A data graph contains a ChangeSummary that can be used to access the change history for any DataObject in the graph. Typically the ChangeSummary is empty when a data graph is returned from a DAS. If the client of the DAS makes modifications that change the state of the DataObjects, including creation and deletion, then a summary of changes can be recorded in the ChangeSummary.

If a client sends a modified data graph to a DAS, (the original DAS or a different one), then the DAS will check the data graph for errors. These errors include lack of closure of the data graph, values outside the lower and upper bounds of a property, choices spanning several properties or DataObjects, deferred constraints, or any restrictions specific to the DAS (for example, XML Schema specific validations). Closure means that any DataObject references, made within the graph of DataObjects, point to a DataObject that is in the graph. Usually, the DAS will report update problems by throwing exceptions.

It is possible that a data graph does not have closure, temporarily, while the contained DataObjects are being modified by an end user, through the DataObject interface. However, after all user operations are completed the data graph should be restored to closure. A DAS should operate only on data graphs with closure.

Java API

The SDO API is made up of the following interfaces that relate to instance data:

- DataObject – A business data object.
- DataGraph – An envelope for a graph of DataObjects.
- ChangeSummary – Summary of changes to the DataObjects in a DataGraph. Sequence - A sequence of settings.

SDO also contains a minimal metadata API that can be used for introspecting the model of DataObjects:

- Type – The Type of a DataObject or Property.
- Property - A Property of a DataObject.

Finally, SDO has a number of helper interfaces and classes:

- DataFactory - For creation of data objects
- CopyHelper - Shallow and deep copy
- EqualityHelper - Shallow and deep equality
- XMLHelper - Serialization to XML
- XMLDocument - Serialization to XML
- XSDHelper - Loading XSDs
- DataHelper
- HelperProvider

The APIs are shown in figure 3 below.

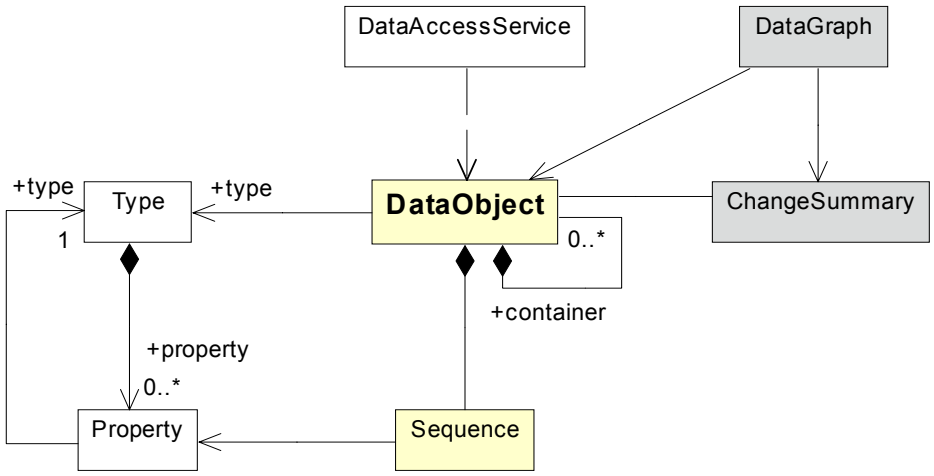


Figure 3: DataGraph APIs

DataObject

DataObjects represent business data. They hold their data in properties.

The DataObject interface is designed to make programming easier because it provides access to business data of all the common types and access patterns, such as name, index, and path.

The DataObject interface includes methods that:

- Get and set the properties of a DataObject.
- Query whether a Property is set.
- Create a new instance of a contained DataObject.
- Delete a DataObject from its container.
- Detach a DataObject from its container.
- Get the container of a DataObject and the containing property.
- Get the root DataObject.
- Get the DataGraph to which a DataObject belongs.
- Get the DataObject's Type.
- Get the DataObject's Sequence (if present).
- Get the DataObject's additional Properties (if present).

For many applications that do not use generated code, the DataObject interface is the only part of SDO that is used to write applications. For many applications that use generated code, the generated interfaces themselves are what is used. The other parts of SDO are primarily use-as-you-go.

DataObject Concepts

DataObjects can be thought of as falling into the following categories. The *open* and *sequenced* concepts can be used independently or together.

1. **Basic**. A DataObject is similar to a JavaBean with a field for each Property. The set of allowed Properties is defined by getType().getProperties(). Values are accessed through get(property). Order within Properties is maintained but not across Properties.
2. **Open.** A DataObject is similar to a JavaBean plus it has tolerance for additional Properties. In XML this is equivalent to open (wildcard) content. It is similar to a JavaBean with an extra Map to hold the new Properties. The extra Properties are not part of getType().getProperties(). The Properties actually set in a specific DataObject are available through getInstanceProperties(). Values are accessed through get(property). Order within Properties is maintained but not across Properties.
3. **Sequenced.** A DataObject is similar to a JavaBean plus it has order within and across Properties. In XML this is equivalent to a DOM. When using XML, a Sequence (see "Sequence" on page 33) represents the order of all the XML elements in the DataObject. Values are available through get(property) but order across Properties is maintained through the Sequence interface. getSequence() returns a Sequence of the XML elements

for the case of XML. XML Attributes do not have the concept of order and are accessed through get(property).

DataObject Values and Properties

DataObjects have data values assigned to Properties. For example, a purchase order DataObject could have the value 2005-06-30 assigned to the orderDate property. Values for the orderDate property can be returned or changed using the get("orderDate") and set("orderDate") accessors on the DataObject. When code is generated, values can also be accessed through getOrderDate() and setOrderDate() methods on a PurchaseOrder interface.

On the DataObject interface, values can be accessed using the name of the property with get(String path), with the index of the property, or directly with a Property object. The get(String path) methods on DataObject work with the alias names as well as the property names names in the path. The path can be just the name of the property, or it can be a path expression based on a subset of XPath.

Type Conversion

Sometimes the Type of a Property is different than the most convenient type for use in an application program. For example, when displaying a integer quantity in a user interface, the string representation is more useful than the int. The method getString("quantity") for accessing an int quantity property conveniently returns the value as a String. This simplifies a common task in many applications.

When a DataObject's typed accessors get<T>() and set<T>() are invoked, a type conversion is necessary if the value is not already an instance of the requested type T. Type conversion is automatically done by a DataObject implementation. An implementation of SDO is expected to convert between any data type and the set defined in DataObject, with possible loss of information. The supported data type set is defined in the SDO DataTypes section. These types include:

- Java primitives
- object wrappers of Java primitives
- String
- Date and time types
- URI
- byte[]
- BigDecimal
- BigInteger

Conversions are specified in Java [6] and the DataHelper. The supported conversions are specified in "DataType Conversions" on page 145.

Many-valued DataObject Properties

A Property can have one or many values. If a Property is many-valued then property.many is true and get(property) always returns a List.

DataObject methods with a return type of List, on the DataObject interface or generated, return empty lists rather than null when there is no value. Returned Lists actively represent any changes to the DataObject's values.

The getList(property) accessor is especially convenient for many-valued properties. If property.many is true then set(property, value) and setList(property, value) require that "value" be a java.util.Collection and List respectively. These methods are equivalent to getList(property).clear() followed by getList(property).addAll(value).

For many-valued Properties, get() and getList() return a List containing the current values. Updates through the List interface operate on the current values of the DataObject immediately. Each access to get() or getList() returns the same List object.

Determining whether a Property is Set

For many-valued properties, isSet(property) returns:

- True, if the List is not empty.
- False, if the List is empty.

For single-valued properties, isSet(property) returns:

- False, if the Property has not been set(), or has been unset().
- True, if the current value is not the Property's default.
- For the remaining cases the implementation may decide between either of the following policies:
  - Any call to set() without a call to unset() will cause isSet() to return true. After set(property, property.getDefault()), isSet(property) returns true.
  - The current value is compared to the default value and isSet() returns true when they differ. After set(property, property.getDefault()), isSet(property) returns false.

The unset(property) accessors can be thought of as clearing out a single property, so that isSet(property) returns false and get(property) returns the default. The delete() method unsets all the DataObject's properties except for those marked read-only. After unset(), get(property) returns the default; which in the case of a many-valued Property is an empty List.

Note that attempts to modify read-only properties (using set, unset or delete) cause an exception.

Containment

DataObjects in a data graph are arranged in a *tree* structure. One DataObject forms the *root* of the tree and the other DataObjects make up the nodes of the tree.

The tree structure is created using *containment references* which start at the root DataObject. The root DataObject refers to other DataObjects, which can refer to further DataObjects. Each DataObject in the data graph, except for the root DataObject, must have a containment reference from another node in the tree. Each DataObject in the graph keeps track of the location of its containment reference.

It is possible for a data graph to have non-containment references. These are references to DataObjects which are part of the same data graph, (the referenced DataObjects must be part of the same tree), but these references do not affect the tree structure of the data graph.

Both containment and non-containment references are Properties of a DataObject. The Type of the Properties is any DataObject Type.

Whether a particular DataObject reference Property is a containment reference or a non-containment reference is defined by the data model for the data graph, for example the XSD which defines the data types for an XML document. This cannot be changed once the data model has been defined. You can query whether a particular reference is a containment reference accessing property.containment.

A container DataObject is one that contains other DataObjects. A DataObject can have a maximum of one container DataObject. If a DataObject has no container, it is considered to be a root DataObject.

Simple navigation, up and down the DataObject containment tree, is provided by getContainer() and getContainmentProperty(). The getContainer() method returns the parent DataObject and the getContainmentProperty() method returns the Property of the container that contains this object. A DataObject can be removed from its container, without making any other changes, using the detach() method.

Containment is managed. When a DataObject is set or added to a containment Property, it is removed from any previous containment Property. Containment cannot have cycles. If a set or add would produce a containment cycle, an exception is thrown.

Creating and Deleting DataObjects

The create methods create a DataObject of the Type of the Property, or the Type specified in the arguments, and add the created object to the Property specified. If the DataObject's Type is a sequenced type (that is, if getType().isSequenced() is true) then the created DataObject is put at the end of the Sequence. If the Property is single-valued, the Property is set to the created object.

If the Property is multi-valued, the created object is added as the last object. Only containment properties may be specified for creation. A created object begins with all its properties unset.

The delete() method unsets all the DataObject's non-readonly properties. The delete() method will also remove the DataObject from its containing DataObject if the containment Property is not read-only. All DataObjects recursively contained by containment properties will also be deleted.

If other DataObjects have one-way, non-containment properties that refer to deleted DataObjects, then these references are not modified. However, these properties can need changing to other values, in order to restore closure to the data graph. A deleted DataObject can be used again, have its values set, and be added into the data graph again.

Sequenced DataObjects

A DataObject can be of a sequenced or unsequenced type (see Sequence). The getType().isSequenced() method tells you whether the DataObject's Type is sequenced or not.

If a DataObject's Type is sequenced then getSequence() returns that Sequence, otherwise getSequence() returns null.

The Sequence of a DataObject corresponds to the XML elements representing the values of its properties. Updates through DataObject, and the Lists or Sequences returned from DataObject, operate on the same data.

Returned Sequences actively represent any changes to the DataObject's values.

Open Content DataObject Properties

DataObjects can have two kinds of Properties:

1. Those specified by their Type (see Type)
2. Those not specified by their Type. These additional properties are called *open content*.

Properties which are specific to a DataObject's Type are returned in a List by getType().getProperties().

Open Content DataObject Properties

DataObjects can have Properties beyond those specified by their Type when either:

1. Handling XML open or mixed content.
2. Encountering new Properties dynamically.

Open content Properties are allowed only when Type.open is true. Some Types set open to false so they do not have to accept additional Properties.

A Property is from open content if it appears in getInstanceProperties() but not in getType().getProperties(). If a Property is from open content then isSet(property) must be true.

Properties with DataType Types may return different objects as long as equals() is true. For mutable data values (Date and List of Strings for example), modification of those values directly is implementation dependent.

All Properties currently used in a DataObject are returned, in a read-only List, when you invoke getInstanceProperties(). This includes properties that are open content. The order of the Properties begins with all the getType().getProperties() whether set or not; the order of the remaining Properties is determined by the implementation. Each invocation of getInstanceProperties() will return the same List object, unless the DataObject is updated so that the contents of the List change.

The property name can be used to find the corresponding Property active on the DataObject within the instance properties by calling getProperty().

In order to set an open content value when that Property is not set (it does not appear in getInstanceProperties()), a set or create accessor on DataObject, or add on List or Sequence, with a Property parameter is used, typically found by accessing the TypeHelper or XSDHelper. An example of creating open content is found in the "Creating open content XML documents" on page 135.

All open content properties in getInstanceProperties() will have isSet(property) return true.

Property Indices

When a DataObject has multiple Properties, each of the Properties can be referenced by an index number, starting at 0 for the first Property.

The Property index used in get(int property), is the position in the List returned by getInstanceProperties().

Using index parameter accessors for open content is **not** recommended if the data is being modified, unless the index is used in coordination with getInstanceProperties(). This is because the index of properties for open content in getInstanceProperties() can change, if the values of several open content properties are set and unset repeatedly.

The following example is acceptable because the index is used in coordination with getInstanceProperties(). Note that DataObjects are not synchronized so the user should not have updates going on at the same time as reads. This example shows a common pattern, looping through all instance properties and printing the property name and value:

```
for (int i=0; i<myDo.getInstanceProperties().size(); i++)
{
  Property p = (Property) myDo.getInstanceProperties().get(i);
  System.out.println(p.getName()+"="+myDo.getString(i));
}
```

Names and alias names for Properties earlier in getInstanceProperties() take precedence over those with a higher index, meaning that open content Properties can have their name hidden by names defined in the Type's Properties since those Properties are at the beginning of the list. The order of precedence is the order in getInstanceProperties().

In the event of a duplicate name, the open content Property can be accessed through its alias name if that does not conflict with any names, or alias names, in the previous Properties.

Current State for a DataObject

The *current state* for a DataObject are all the values that distinguish it from a newly created object from the DataFactory, since newly created objects from a DataFactory have no properties set and no container. The current state for a DataObject are all the properties in getInstanceProperties() where isSet() returns true. The container and containment property are part of the state of the containing DataObject. This program prints the current state of the DataObject myDO.

```
for (int i=0; i<myDo.getInstanceProperties().size(); i++)
{
  Property p = (Property) myDo.getInstanceProperties().get(i);
  if (myDo.isSet(p))
```

```
        {
          System.out.println(p.getName()+"="+myDo.getString(i));
        }
      }
```

DataObject Interface

```
public interface DataObject extends Serializable
{
  Object get(String path);
  void set(String path, Object value);
  boolean isSet(String path);
  void unset(String path);

  boolean getBoolean(String path);
  byte getByte(String path);
  char getChar(String path);
  double getDouble(String path);
  float getFloat(String path);
  int getInt(String path);
  long getLong(String path);
  short getShort(String path);
  byte[] getBytes(String path);
  BigDecimal getBigDecimal(String path);
  BigInteger getBigInteger(String path);
  DataObject getDataObject(String path);
  Date getDate(String path);
  String getString(String path);
  List getList(String path);
  Sequence getSequence(String path);

  void setBoolean(String path, boolean value);
  void setByte(String path, byte value);
  void setChar(String path, char value);
  void setDouble(String path, double value);
  void setFloat(String path, float value);
  void setInt(String path, int value);
  void setLong(String path, long value);
  void setShort(String path, short value);
  void setBytes(String path, byte[] value);
  void setBigDecimal(String path, BigDecimal value);
  void setBigInteger(String path, BigInteger value);
  void setDataObject(String path, DataObject value);
  void setDate(String path, Date value);
  void setString(String path, String value);
  void setList(String path, List value);
```

```
Object get(int propertyIndex);
void set(int propertyIndex, Object value);
boolean isSet(int propertyIndex);
void unset(int propertyIndex);

boolean getBoolean(int propertyIndex);
byte getByte(int propertyIndex);
char getChar(int propertyIndex);
double getDouble(int propertyIndex);
float getFloat(int propertyIndex);
int getInt(int propertyIndex);
long getLong(int propertyIndex);
short getShort(int propertyIndex);
byte[] getBytes(int propertyIndex);
BigDecimal getBigDecimal(int propertyIndex);
BigInteger getBigInteger(int propertyIndex);
DataObject getDataObject(int propertyIndex);
Date getDate(int propertyIndex);
String getString(int propertyIndex);
List getList(int propertyIndex);
Sequence getSequence(int propertyIndex);

void setBoolean(int propertyIndex, boolean value);
void setByte(int propertyIndex, byte value);
void setChar(int propertyIndex, char value);
void setDouble(int propertyIndex, double value);
void setFloat(int propertyIndex, float value);
void setInt(int propertyIndex, int value);
void setLong(int propertyIndex, long value);
void setShort(int propertyIndex, short value);
void setBytes(int propertyIndex, byte[] value);
void setBigDecimal(int propertyIndex, BigDecimal value);
void setBigInteger(int propertyIndex, BigInteger value);
void setDataObject(int propertyIndex, DataObject value);
void setDate(int propertyIndex, Date value);
void setString(int propertyIndex, String value);
void setList(int propertyIndex, List value);

Object get(Property property);
void set(Property property, Object value);
boolean isSet(Property property);
void unset(Property property);

boolean getBoolean(Property property);
byte getByte(Property property);
char getChar(Property property);
double getDouble(Property property);
```

```
  float getFloat(Property property);
  int getInt(Property property);
  long getLong(Property property);
  short getShort(Property property);
  byte[] getBytes(Property property);
  BigDecimal getBigDecimal(Property property);
  BigInteger getBigInteger(Property property);
  DataObject getDataObject(Property property);
  Date getDate(Property property);
  String getString(Property property);
  List getList(Property property);
  Sequence getSequence(Property property);

  void setBoolean(Property property, boolean value);
  void setByte(Property property, byte value);
  void setChar(Property property, char value);
  void setDouble(Property property, double value);
  void setFloat(Property property, float value);
  void setInt(Property property, int value);
  void setLong(Property property, long value);
  void setShort(Property property, short value);
  void setBytes(Property property, byte[] value);
  void setBigDecimal(Property property, BigDecimal value);
  void setBigInteger(Property property, BigInteger value);
  void setDataObject(Property property, DataObject value);
  void setDate(Property property, Date value);
  void setString(Property property, String value);
  void setList(Property property, List value);

  DataObject createDataObject(String propertyName);
  DataObject createDataObject(int propertyIndex);
  DataObject createDataObject(Property property);
  DataObject createDataObject(String propertyName, String namespaceURI, String
typeName);
  DataObject createDataObject(int propertyIndex, String namespaceURI, String
typeName);
  DataObject createDataObject(Property property, Type type);

  void delete();
  void detach();

  DataObject getContainer();
  Property getContainmentProperty();

  DataObject getRootDataObject();
  DataGraph getDataGraph();

  Type getType();
```

```
  Sequence getSequence();

  List getInstanceProperties();

}
```

A Java implementation of DataObject must not override the methods defined in java.lang.Object except for toString().

DataObject Accessor Exceptions

The following exceptions are thrown on DataObject accessors. These exceptions are all standard Java runtime exceptions so programs do not need try/catch blocks to program with the DataObject interface. The content of the exception is a String that describes the problem.

SDO specifies minimum functionality for implementations. An implementation may provide additional function so that valid results would be returned where this specification would produce an error, provided that the functionality is a strict superset and all valid uses of the SDO specification operate correctly.

The get(String path) method will return null instead of throwing an exception for error conditions. This avoids the need for defensive programming and helps simple programs access data that has a flexible structure.

get<T>(String path) and set<T>(String path) only throw a ClassCastException if it is impossible to convert between the actual and expected types.

Open content DataObjects will not throw exceptions for accessing properties which are not set on the DataObject.

Validation of Facets and Constraints

XML elements can have facets, that is, restrictions. If the value set on a Property does not meet a facet or constraint, such as an XSD range restriction, the accessor may throw an IllegalArgumentException. However, implementations are not required to throw exceptions because it can be more practical to perform validation at a later time.

Validation that occurs during the execution of an accessor method is called *immediate* validation. Validation that is externally triggered is called *deferred* validation. In general, deferred validation is more efficient because checking is typically invoked once, after all values are set. Most constraints can only be enforced with deferred validation because more than a single property value is being validated. Underflow constraints (that is properties that must be assigned values for valid input to an application) are always deferred when building new DataObjects. SDO leaves it to implementations, applications, and services to determine when and how validation should be performed. Deferred validation is defined by services which perform validation on their input parameters, for example before the service makes updates to a database. Deferred validation does not occur through the DataObject APIs.

If an exception is thrown, no change to the DataObject takes place and therefore there is no change to any ChangeSummary.

Condition	Exception
For Types without open content (open=false), Property is not a member of getInstanceProperties() in get<T>(Property property) or get<T>(int propertyIndex). • getInstanceProperties().contains(property) == false • propertyIndex < 0 or >= getInstanceProperties().size() o Example: get(null) o Example: get(-1) o Example: isSet(property) **NOTE**: get<T>(String path) does not throw exceptions.	IllegalArgumentException
Index out of range on a multi-valued Property (defined by the List interface) • index < 0 or >= getList(Property property).size() o Example: getList(employee).get(-1) o Example: getList(employee).get(1000) where there are less than 1000 values	IndexOutOfBoundsException

Modification of a read-only property • Example: set(employeeNumber, 123) where employeeNumber.isReadOnly() == true • Example: unset(employeeNumber) where employeeNumber.isReadOnly() == true • Example: getList(employees).remove(anEmployee) or • Example: anEmployee.detach() or • Example: anEmployee.delete() where employees.isReadOnly()==true and anEmployee.getContainmentProperty()==employees.	UnsupportedOperationException
Cannot convert between value and requested Type • Example: getDate(property) where property.Type is float • Example: getList(property) where property.many == false and property.type.instanceClass is not List.	ClassCastException
Mixing single-valued and multi-valued Property access • Example: getList(property) where property.many == false • Example: getInt(property) where property.many == true	ClassCastException
Circular containment • Example: a.set("child", b); b.set("child", c); c.set("child", a) where child is a containment Property.	IllegalArgumentException

DataGraph

A DataGraph is an optional envelope for a graph of DataObjects with a ChangeSummary.

To obtain the same functionality as the DataGraph with DataObjects alone, DataObjects may be defined using the SDO DataGraph XSD.

A ChangeSummary may be used directly with DataObjects as explained in the ChangeSummary section.

The DataGraph has methods to:

- return the root DataObject
- create a rootDataObject if one does not yet exist.
- return the change summary
- look up a type by uri and name similar to the TypeHelper.

DataGraph Interface

```
public interface DataGraph extends Serializable
{
  DataObject getRootObject();

  DataObject createRootObject(String namespaceURI, String typeName);
  DataObject createRootObject(Type type);

  ChangeSummary getChangeSummary();

  Type getType(String uri, String typeName);
}
```

Creating DataGraphs

A DataGraph is created by a DAS, which returns either an empty DataGraph, or a DataGraph filled with DataObjects. An empty DataGraph can have a root assigned by the createRootObject() methods. However, if a previous root DataObject exists then an IllegalStateException is thrown.

The DAS is also responsible for the creation of the metadata (that is, the model) used by the DataObjects and DataGraph. For example, a DAS for XML data could construct the model from the XSD for the XML.

Modifying DataGraphs

In order to change a DataGraph a program needs to access the root DataObject, using the getRootObject() method. All other DataObjects in the tree are accessible by recursively traversing the containment references of the root DataObject.

Accessing Types

A Type can be accessed using getType(String uri, String typeName) or through the TypeHelper. This returns a Type with the appropriate URI and name. The convention for getType(), and all methods with a URI parameter, is that the URI is a logical name such as a targetNamespace.

The implementation of DataGraph, TypeHelper, and DataObject is responsible for accessing the physical resource that contains the requested metadata. The physical resource can be a local copy or a resource on a network.

The configuration information necessary to provide this logical to physical mapping, is via implementation-specific configuration files.

If metadata is unavailable, then an implementation-specific exception occurs.

ChangeSummary

A ChangeSummary provides access to change history information for the DataObjects in a data graph.
A change history covers any modifications that have been made to a data graph starting from the point when logging was activated. If logging is no longer active, the log includes only changes that were made up to the point when logging was deactivated. Otherwise, it includes all changes up to the point at which the ChangeSummary is being interrogated. Although change information is only gathered when logging is on, you can query change information whether logging is on or off. All of the information returned is read-only.

This interface has methods that:

- Activate and deactivate logging.
- Restore a tree of DataObjects to the state it was in when logging began; and clear the log.
- Query the logging status.
- Get the DataGraph to which the ChangeSummary belongs.
- Get the ChangeSummary's root DataObject.
- Get the changed DataObjects.
- Indicate whether a DataObject has been created, deleted or changed.
- Get the container DataObject at the point when logging began.
- Get a DataObject's containment Property at the point when logging began.
- Get a DataObject's Sequence at the point when logging began.
- Get a specific old value.
- Get a List of old values.

Starting and Stopping Change Logging

beginLogging() clears the ChangeSummary's List of changed DataObjects and starts change logging. endLogging() stops change logging. undoChanges() restores the tree of DataObjects to its state when logging began. undoChanges() also clears the log, but does not affect isLogging().

NOTE: The beginLogging(), endLogging() and undoChanges() methods are intended primarily for the use of service implementations since services define how the processing of a ChangeSummary relates to external resources. Making changes that are not captured in the ChangeSummary may cause services that drive updates from a ChangeSummary to act on incomplete information.

Scope

The scope of a ChangeSummary is defined as the containment tree of DataObjects from the ChangeSummary root. The ChangeSummary root is the DataObject from which all changes are

tracked. The ChangeSummary root is retuned by getRootObject(). This object is one of the following:

- The DataObject with the ChangeSummary as a value.
- The root DataObject of a DataGraph.

Old Values

A List of old values can be retrieved using the getOldValues(DataObject dataObject) method. The order of old values returned is implementation dependent. For a deleted DataObject, the old values List contains all the properties of the DataObject. For a DataObject that has been modified, the old values List consists of the modified properties only. For a DataObject that has not been deleted or modified, the List of old values is empty.

Old values are expressed as ChangeSummary.Setting objects (ChangeSummary.Setting is an inner interface of ChangeSummary). Each ChangeSummary.Setting has a Property and a value, along with a flag to indicate whether or not the Property is set.

getOldValue(DataObject dataObject, Property property) returns a ChangeSummary.Setting for the specified Property, if the DataObject was deleted or modified. Otherwise, it returns null. If the setting.isSet() of the old value is false, the old value does not have meaning.

Sequenced DataObject

getOldSequence(DataObject dataObject) returns the entire value of a DataObject's Sequence, at the point when logging began. This return value can be null. If DataObject.getSequence() returns null then getOldSequence(DataObject dataObject) will return null.

Serialization and Deserialization

When a ChangeSummary is deserialized, the logging state will be on if a <changeSummary> element is present in the XML unless the changeSummary marks logging as off. A serializer must produce a <changeSummary> element in the XML if either of the following conditions applies:

1. Changes have been logged (getChangedDataObjects().size() > 0).
2. No changes have been logged **but** isLogging() is true at the time of serialization. In this case, an empty <changeSummary/> or <changeSummary logging="true"/> element must be produced.

The state of logging is recorded in the logging attribute of the changeSummary element.

The serialization of a ChangeSummary includes enough information to reconstruct the original information of the DataObjects, at the point when logging was turned on. The create attribute labels DataObjects currently in the data graph that were not present when logging started, and the

delete attribute labels objects contained in the change summary that are no longer in the data graph. Labels are either IDs, if available, or sdo path expressions.

The contents of a ChangeSummary element are either deep copies of the objects at the point they were deleted, or a prototype of an object that has had only data type changes, with values for the properties that have changed value.

Associating ChangeSummaries with DataObjects

There are two possible ways to associate DataObjects and ChangeSummaries:

1. DataGraphs can get a ChangeSummary using the getChangeSummary() method.
   - This is used when a ChangeSummary is external to the DataObject tree. The ChangeSummary tracks changes on the tree of DataObjects starting with the root DataObject available through DataGraph's getRootObject().
2. The Type of a DataObject can include a Property for containing a ChangeSummary.
   - This is used when a ChangeSummary is part of a DataObject tree, for example when a root DataObject is a message header that contains both a message body of DataObjects and a ChangeSummary. The ChangeSummary tracks changes on the tree of DataObjects starting with the DataObject that contains the ChangeSummary.

ChangeSummary Interface

The ChangeSummary interface provides methods to

- check to status of logging, or turn logging on and off
- undo all the changes in the log to the point when logging began
- return the root DataObject and DataGraph
- return DataObjects that have been modified, created, deleted
- identify what kind of change (modifed, created, deleted) has occured
- return the old values for changed and deleted DataObjects

```
public interface ChangeSummary
{
  void beginLogging();
  void endLogging();
  boolean isLogging();

  void undoChanges();

  DataGraph getDataGraph();
  DataObject getRootObject();

  List /*DataObject*/ getChangedDataObjects();
  boolean isCreated(DataObject dataObject);
  boolean isDeleted(DataObject dataObject);
  boolean isModified(DataObject dataObject);
  DataObject getOldContainer(DataObject dataObject);
```

```
  Property getOldContainmentProperty(DataObject dataObject);
  Sequence getOldSequence(DataObject dataObject);

  public interface Setting
  {
    Object getValue();
    Property getProperty();
    boolean isSet();
  }

  Setting getOldValue(DataObject DataObject, Property property);
  List /*Setting*/ getOldValues(DataObject dataObject);
}
```

Sequence

A Sequence is an ordered collection of settings. Each entry in a Sequence has an index.

The key point about a Sequence is that the order of settings is preserved, even across different properties. So, if Property A is updated, then Property B is updated and finally Property A is updated again, a Sequence will reflect this.

Each setting is a property and a value. There are shortcuts for using text with the SDO text Property.

Unstructured Text

Unstructured text can be added to a Sequence using the SDO text Property. The add(String text) method adds a new entry, with the SDO text Property, to the end of the Sequence. The add(int index, String text) method adds a new entry, with the SDO text Property, at the given index.

Using Sequences

Sequences are used when dealing with semi-structured business data, for example mixed text XML elements. Suppose that a Sequence has two many-valued properties, say “numbers” (a property of type int) and “letters” (a property of type String). Also, suppose that the Sequence is initialized as follows:

1. The value 1 is added to the numbers property.
2. The String “annotation text” is added to the Sequence.
3. The value “A” is added to the letters property
4. The value 2 is added to the numbers property.
5. The value “B” is added to the letters property.

At the end of this initialization, the Sequence will contain the settings:

{<numbers, 1>, <text, ”annotation text”>, <letters, ”A”>, <numbers, 2>, <letters, ”B”>}

The numbers property will be set to {1, 2} and the letters property will be set to {“A”, ”B”}, but the order of the settings across numbers and letters will not be available through accessors other than the sequence.

Comparing Sequences with DataObjects

The way in which a DataObject keeps track of the order of properties and values is quite different from the way in which a Sequence keeps track of the order.

The order in which different properties are added to a DataObject is not preserved. In the case of a many-valued Property, the order in which different values are added to that one Property is preserved, but when values are added to two different Properties, there is no way of knowing which Property was set first. In a Sequence, the order of the settings across properties is preserved.

The same properties that appear in a Sequence are also available through a DataObject, but without preserving the order across properties.

Note that if a DataObject's Type is a sequenced type (that is, if getType().isSequenced() is true) then a DataObject will have a Sequence.

Sequence Methods

- The size() method returns the number of entries in the Sequence.
- The getProperty(int index) accessor returns the Property at the given index.
- The getValue(int index) accessor returns the value at the given index.
- The setValue(int index, Object value) accessor updates the value at the given index and maintains sequence positions.
- The boolean add() accessors add to the end of the sequence.
- The add(int index, String text) accessor adds unstructured text, at the given index.
- The add(String text) accessor adds unstructured text at the end of the sequence.
- The other add(int index) accessors add to the specified position in a sequence and, like java.util.List, shift entries at later positions upwards.
- The remove() method removes the entry at the specified index and shifts all later positions down.
- The move() method moves the entry at the fromIndex to the toIndex, shifting entries later than fromIndex down, and entries after toIndex up.

To create DataObjects at the end of a Sequence, the create() methods on DataObject may be used.

Sequence Interface

```
public interface Sequence
```

```
{
  int size();

  Property getProperty(int index);
  Object getValue(int index);

  Object setValue(int index, Object value);

  boolean add(String propertyName, Object value);
  boolean add(int propertyIndex, Object value);
  boolean add(Property property, Object value);
  void add(int index, String propertyName, Object value);
  void add(int index, int propertyIndex, Object value);
  void add(int index, Property property, Object value);
  void add(int index, String text);
  void add(String text);

  void remove(int index);
  void move(int toIndex, int fromIndex);
}
```

Type

The Type interface represents a common view of the model of a DataObject, or of a data type.

The concept of a data type is shared by most programming languages and data modeling languages; and SDO Types can be compared with other data types. An SDO Type has a set of Property objects, unless it represents a simple data type.

Mapping SDO Types to Programming and Data Modeling Languages

Java, C++, UML or EMOF Class

- Class can be represented by an SDO Type.
- Each field of the Class can be represented by an SDO Property.

XML Schema

- Complex and simple types can be represented by SDO Types.
- Elements and attributes can be represented by SDO Properties.

C Struct

- C Struct can be represented by an SDO Type
- Each field of the Struct can be represented by an SDO Property.

Relational database

- Table can be represented by an SDO Type.
- Column can be represented by an SDO Property.

All of these domains share certain concepts, a small subset of which is represented in the SDO Type and Property interfaces. These interfaces are useful for DataObject programmers who need to introspect the shape or nature of data at runtime.

More complete metamodel APIs (for example, XML Schema or EMOF) representing all the information of a particular domain are outside the scope of this specification.

Type Contents

A Type will always have:

- Name - A String that must be unique among the Types that belong to the same URI.
- URI - The logical URI of a package or a target namespace, depending upon your perspective.
- Boolean fields indicating if the type is open, abstract, sequenced, or a data type.

A Type can have:

- Properties - a list of Property objects defined by this Type. Types corresponding to simple data types define no properties.
- Instance Class - the java.lang.Class used to implement the SDO Type.
    - If DataType is true then a Type **must** have an Instance Class. Example classes are: java.lang.Integer and java.lang.String.
    - If DataType is false, and generated code is used, then an Instance Class is optional. Examples classes might be: PurchaseOrder and Customer.
- Aliases - Strings containing additional names. Alias Names must be unique within a URI. All methods that operate on a Type by name also accept alias names. For example, a Type might be assigned an alias name for the domains it is used in: an XML Schema name "PurchaseOrderType", a Java name "PurchaseOrder" and a database table name "PRCHORDR".

Name Uniqueness

Type names and Type alias names are all unique within a URI. Property names and Property alias names are all unique within a Type and any base Types.

SDO Data Types

SDO defines Types for the common data types supported in SDO, enabling more consistency in defining the Types and Properties used by services. Refer to "Standard SDO Types" on page 71 for more details.

Multiple Inheritance

Type supports multiple inheritance by allowing multiple base types. When multiple inheritance is used, the order of Properties in getProperties() may differ between a Type and the order in the base Types.

Type Methods

- getName() returns the Type Name.
- getURI returns the Type URI.
- getInstanceClass() returns the Class used to implement the SDO Type.
- isInstance(Object object) returns true if the specified object is an instance of this Type.
- isDataType() returns true if this Type specifies DataTypes and returns false for DataObjects.
- isSequenced() returns true if this Type specifies Sequenced DataObjects. When true, a DataObject can return a Sequence.
- isOpen() returns true if this Type allows open content. If false, then dataObject.getInstanceProperties() must be the same as dataObject.getType().getProperties() for any DataObject of this Type.
- isAbstract() returns true if this Type is abstract, that is cannot be instantiated. Abstract types cannot be used in DataObject or DataFactory create methods. Abstract types typically serve as the base Types for instantiable Types.
- getBaseTypes() returns a List of base Types for this Type. The list is empty •if there are no base Types. XSD <extension>, <restriction>, and the •Java "extends" keyword are mapped to this List of base Types.
- getAliasNames() returns a List of alias names for this Type. The list is empty if there are no Aliases.
- getProperties() returns all Properties for this Type including those declared in the base Types.
- getDeclaredProperties() returns the Properties declared in this Type as opposed to those declared in base Types.
- getProperty(String propertyName) returns a particular Property or null if there is no property with the given name.

Type Interface

```
public interface Type
{
  String getName();
  String getURI();

  Class getInstanceClass();
  boolean isInstance(Object object);

  boolean isDataType();
  boolean isSequenced();
```

```
  boolean isOpen();
  boolean isAbstract();

  List /*Type*/ getBaseTypes();
  List /*String*/ getAliasNames();

  List /*Property*/ getProperties();
  List /*Property*/ getDeclaredProperties();
  Property getProperty(String propertyName);
}
```

Property

A DataObject is made up of Property values.

A Property has:

- Name - a String that is typically unique among the Properties that belong to the DataObject.
- Type - the Type of this Property. A Property whose Type is for DataObjects is sometimes called a reference; otherwise it is called an attribute.
- Containment - whether the property is a containment property. A property with containment true is called a ***containment property***.
- Many - whether the property is single-valued or many-valued.
- ReadOnly - whether the property may be modified through the DataObject or generated API.
- Alias names - alternative names that must be unique within the Type. A Property might be assigned an alias name for the domains it is used in, such as an XMLSchema name "firstName", a Java name "first_name", and a database column name, "FRSTNAME". All Property names and all alias names for Properties must be unique for all Properties in Type.getProperties().
- Default value.
- Numeric index within the Property's Type.

Property Methods

- getName() returns the Property Name.
- getType() returns the Property Type.
- isMany() returns true if the Property is many-valued, or false if the Property is single-valued.
- isContainment() returns true if the Property is a containment reference.
- isReadOnly() returns true if values for this Property cannot be modified using the SDO APIs.
- getContainingType() returns the Type that declares this Property.
- getAliasNames() returns a list of alias names for this Property.
- getOpposite() returns the opposite Property, if the Property is bi-directional, otherwise returns null.

- getDefault() returns the default value (as an Object).

Property Index

Each Type assigns a unique index to each Property that belongs to a DataObject. The index can be accessed in the List returned by Type.getProperties().

Containment

In the case of a reference, a Property may be either a containment or non-containment reference. In EMOF, the term containment reference is called composite. In XML, elements with complex types are mapped to containment properties.

A Property with containment true is called a containment property. Containment properties show the parent-child relationships in a tree of DataObjects.

Read-Only Properties

Read-Only Properties **cannot** be modified using the SDO APIs. When DataObject.delete() is invoked, read-only Properties are not changed. Any attempt to alter read-only Properties using DataObject.set(Property property, Object value) or unset() results in an exception.

Read-Only Properties **can** be modified by a service using implementation-specific means. For Example, suppose a relational database service returns a DataObject in which the customerName Property is marked read-only. If the DataObject is passed back to the service, the value of the customerName could be updated by the service to the current value in the database.

Property Interface

```
public interface Property
{
  String getName();
  Type getType();

  boolean isMany();
  boolean isContainment();
  boolean isReadOnly();

  Type getContainingType();

  List /*String*/ getAliasNames();

  Property getOpposite();
```

```
  Object getDefault();
}
```

DataFactory

A DataFactory is a helper for the creation of DataObjects. The created DataObjects are not connected to any other DataObjects. Only Types with DataType false and abstract false may be created.

Default DataFactory

The default DataFactory is available from the INSTANCE field. It is configured in an implementation-specific fashion to determine which Types are available and what instance classes are instantiated.

The default DataFactory uses the default TypeHelper:

- DataFactory.INSTANCE.create(Class) is a shortcut to
    - DataFactory.INSTANCE.create(TypeHelper.INSTANCE.getType(Class))
- DataFactory.INSTANCE.create(String, String) is a shortcut to
    - DataFactory.INSTANCE.create(TypeHelper.INSTANCE.getType(String, String)).

A DataFactory other than the default may have access to a different type helper.

Creating DataObjects

For all create methods:

- Type.dataType and abstract must both be false.
- The Type's getInstanceClass() method returns the same object as the interfaceClass parameter.
- Throw an IllegalArgumentException if the instanceClass does not correspond to a Type this factory can instantiate.
- The created DataObject implements the DataObject interface and the interface specified by the Type.instanceClass, if one exists. There is always an SDO Type and instance relationship and there can also be a Java Class and instance relationship. If there is a Java instance class specified on the Type then both the SDO and the Java relationships hold.
- The created object's getType() will return the Type and the Type.isInstance() will return true for the created object.

create(Class interfaceClass)

- Creates a DataObject that implements both the interfaceClass and DataObject interfaces.
- The interfaceClass is the Java interface that follows the SDO code generation pattern.

- This method only applies to Types that have instance classes.
- The effect of this call is the same as determining the Type for the interfaceClass and calling the create(Type) method.

create(String uri, String typeName)

- Creates a DataObject of the Type specified by typeName with the given package uri.
- The uri and typeName parameters are of the same form as the TyperHelper and DataGraph getType() methods. They uniquely identify a Type from the metadata.
- The effect of this call is the same as determining the Type for the uri and typeName and calling the create(Type) method
- This method applies to Types whether they have instance classes or not. If the Type has an InstanceClass then the returned object will be an instance.

• create(Type type)

- Creates a DataObject of the Type specified.
- This method applies to Types whether they have instance classes or not. If the Type has an instance class then the returned object will be an instance.

NOTE: There is a special case if the Type used in a create() method has a property of type SDO ChangeSummaryType. In this case, the created object will be associated with a new ChangeSummary instance and change logging will be off.

DataFactory Interface

```
public interface DataFactory
{
  DataObject create(Class interfaceClass);
  DataObject create(String uri, String typeName);
  DataObject create(Type type);

  DataFactory INSTANCE = HelperProvider.getDataFactory();
}
```

TypeHelper

A TypeHelper is a helper for looking up Types and for defining new SDO Types, dynamically.

Default TypeHelper

The default TypeHelper is available from the INSTANCE field. It is configured in an implementation-specific fashion to determine which Types are available and what instance classes are known.

When SDO methods have String parameters to specify the name and URI of a Type, the behavior is the same as if they had used TypeHelper getType() with the same parameters. The scope of the types available through any SDO API includes all those available through TypeHelper.INSTANCE.

TypeHelper Methods

- getType(String uri, String typeName) returns the Type specified by typeName with the given uri, or null if not found. This is the helper version of DataGraph.getType(String uri, String typeName) for a given DataGraph.
- getType(Class interfaceClass) returns the Type for this interfaceClass or null if not found.
- define(DataObject type) defines the DataObject as a Type.
- define(List types) defines the list of DataObjects as Types.

TypeHelper Interface

```
public interface TypeHelper
{
  Type getType(String uri, String typeName);
  Type getType(Class interfaceClass);

  Type define(DataObject type);
  List /*Type*/ define(List /*DataObject*/ types);

  TypeHelper INSTANCE = HelperProvider.getTypeHelper();
}
```

Defining SDO Types Dynamically

It is possible to define new SDO Types dynamically using TypeHelper. For example, to define a new Customer Type you could use the TypeHelper as follows:

```
TypeHelper types = TypeHelper.INSTANCE;
Type intType = types.getType("commonj.sdo", "Int");
Type stringType = types.getType("commonj.sdo", "String");

// create a new Type for Customers
DataObject customerType = DataFactory.INSTANCE.create("commonj.sdo", "Type");
customerType.set("uri", "http://example.com/customer");
customerType.set("name", "Customer");
```

```
// create a customer number property
DataObject custNumProperty = customerType.createDataObject("property");
custNumProperty.set("name", "custNum");
custNumProperty.set("type", intType);

// create a first name property
DataObject firstNameProperty = customerType.createDataObject("property");
firstNameProperty.set("name", "firstName");
firstNameProperty.set("type", stringType);

// create a last name property
DataObject lastNameProperty = customerType.createDataObject("property");
lastNameProperty.set("name", "lastName");
lastNameProperty.set("type", stringType);

// now define the Customer type so that customers can be made
types.define(customerType);
```

Using SDO Dynamic Types

To use the dynamically created Customer Type you could do as follows:

```
DataFactory factory = DataFactory.INSTANCE;

DataObject customer1 = factory.create("http://example.com/customer",
"Customer");
customer1.setInt("custNum", 1);
customer1.set("firstName", "John");
customer1.set("lastName", "Adams");

DataObject customer2 = factory.create("http://example.com/customer",
"Customer");
customer2.setInt("custNum", 2);
customer2.set("firstName", "Jeremy");
customer2.set("lastName", "Pavick");
```

CopyHelper

A CopyHelper creates copies of DataObjects.

Default CopyHelper

The default CopyHelper is available from the INSTANCE field.

CopyHelper Methods

- DataObject copyShallow(DataObject dataObject) creates a shallow copy of the dataObject.
- DataObject copy(DataObject dataObject) creates a deep copy of the dataObject tree.

Shallow Copies

copyShallow(DataObject dataObject) creates a new DataObject with the same values as the source dataObject, for each Property where property.type.dataType is true.
If the source's property.type.dataType is false, then that property is unset in the copied DataObject. Read-only properties are copied.

For single-valued Properties:

- copiedDataObject.get(property) <==> dataObject.get(property).

• For many-valued Properties:

- copiedDataObject.getList(property).get(i) <==> dataObject.getList(property).get(i).

• Where <==> means equals() for DataType Properties or the corresponding copied DataObject for DataObject Properties.
• A copied object shares metadata with the source object. For example:

- sourceDataObject.getType() == copiedDataObject.getType().

If a ChangeSummary is part of the source DataObject then the copy has a new, empty ChangeSummary. The logging state of the new ChangeSummary is the same as the source ChangeSummary.

Deep Copies

copy(DataObject dataObject) creates a deep copy of the DataObject tree, that is it copies the dataObject and all its contained DataObjects recursively.

For each Property where property.type.dataType is true, the values of the Properties are copied as in the shallow copy. Read-only properties are copied.

For each Property where property.type.dataType is false, the value is copied if it is a DataObject contained by the source dataObject.

If a DataObject is outside the DataObject tree and the property is bidirectional, then the DataObject is skipped. If a DataObject is outside the DataObject tree and the property is unidirectional, then the same DataObject is referenced.

If a ChangeSummary is part of the copy tree then the new ChangeSummary refers to objects in the new DataObject tree. The logging state is the same as for the source ChangeSummary.

CopyHelper Interface

The CopyHelper has methods to

- create a copy of a DataObject's values with datatype properties
- create a copy of a graph of DataObjects

```
public interface CopyHelper
{
  DataObject copyShallow(DataObject dataObject);
  DataObject copy(DataObject dataObject);

  CopyHelper INSTANCE = HelperProvider.getCopyHelper();
}
```

EqualityHelper

An EqualityHelper compares DataObjects to decide if they are equal.

Default EqualityHelper

The default EqualityHelper is available from the INSTANCE field.

EqualityHelper Methods

- equalShallow(DataObject dataObject1, DataObject dataObject2) returns true if two DataObjects have the same Type, and all their compared Properties are equal.
- equal(DataObject dataObject1, DataObject dataObject2) returns true if two DataObjects are equalShallow(), all their compared Properties are equal, and all reachable DataObjects in their graphs (excluding containers) are equal.

EqualityHelper Interface

The EqualityHelper has methods to

- determine if two DataObjects have the same values for their datatype properties
- determine if two graphs of DataObjects are equal

```
public interface EqualityHelper
{
  boolean equalShallow(DataObject dataObject1, DataObject dataObject2);
  boolean equal(DataObject dataObject1, DataObject dataObject2);

  EqualityHelper INSTANCE = HelperProvider.getEqualityHelper();
}
```

XMLHelper

An XMLHelper converts XML streams to and from graphs of DataObjects.

XMLHelper can be used with or without an XSD. All closed trees of DataObjects are supported, whether or not an XSD was specified. However, the XML will use an XSD if one is used to define the DataObjects.

XMLHelper supports the case where a DataObjects's Types and Properties did not originate in an XSD. It does this by writing XML documents that follow the Generation of XSDs portion of this specification.

Default XMLHelper

The default XMLHelper is available from the INSTANCE field. It is configured in an implementation-specific fashion to determine which Types are available and what instance classes are instantiated.

XMLHelper Methods

- load(String inputString) creates and returns an "XMLDocument" on page 50 from the input String. This method does not perform XSD validation by default.
- load(InputStream inputStream) creates and returns an XMLDocument from the inputStream. The InputStream will be closed after reading. This method does not perform XSD validation by default.
- save(XMLDocument xmlDocument, OutputStream outputStream, Object options) serializes an XMLDocument as an XML document into the outputStream. If the DataObject's Type was defined by an XSD, the serialization will follow the XSD. Otherwise, the serialization will follow the format as if an XSD were generated as defined by the SDO specification.
- save(DataObject dataObject, String rootElementURI, String rootElementName, OutputStream outputStream) saves the DataObject as an XML document with the specified root element.
- save(DataObject dataObject, String rootElementURI, String rootElementName) returns the DataObject saved as an XML document with the specified root element.
- createDocument(DataObject dataObject, String rootElementURI, String rootElementName) creates an XMLDocument with the specified XML rootElement for the DataObject.

Loading and Saving XML Documents

Options can be specified for some load() and save() methods, using the options parameter.

The XMLHelper and XMLDocument do not change the state of the input DataObject and ignore any containers. After load, the root DataObject created does not have a containing DataObject.

When loading XML documents, typically the Types and Properties are already defined, for example from an XSD. If there are no definitions, the XML without Schema to XSD is used. In some situations, the definitions of the Types and Properties have changed relative to the software that has originally written the document, often called *schema evolution*. SDO does not directly address schema evolution, which is an issue broader than SDO, but the general guideline is to use the same URI for compatible XML documents and different URIs for incompatible XML documents.

XML Schemas

Often, it is desirable to validate XML documents with an XSD. To ensure validation, the root element name and URI must correspond to a global element name and target namespace in an XSD.

If an XSD is not being used, for example when the schema types were created dynamically with TypeHelper, then the following is suggested. Choose element names and URIs from the global elements that would be produced by the Generation of XSDs. This improves integration with software that does make use of XSDs.

To enable XML support for DataObjects when XSDs are not used, the following conventions apply to root elements. Although the use of a valid global element is not enforced, it is encouraged whenever an XSD is available.

- When saving the root element, an xsi:type may always be written in the XML to record the root DataObject's Type. If the rootElementURI and rootElementName correspond to a valid global element for that DataObject's Type, then an implementation may suppress the xsi:type.
- When loading the root element, if an xsi:type declaration is found, it is used as the type of the root DataObject. Unless XSD validation is being performed, it is not an error if the rootElementURI and rootElementName do not correspond to a valid global element.
- The root element "commonj.sdo", "dataObject" may be used with any DataObject if xsi:type is also written for the actual DataObject's Type.

To enable XML support for DataObjects when multiple inheritance is used, an additional convention is applied, since XSD cannot support multiple inheritance. The documents will resemble those where single inheritance is used, but will not validate with an XSD because no XSD definition is possible. This convention applies when serializing an element representing a DataObject where the DataObject's Type has more than one Base Type:

- xsi:type is included in the serialization of the DataObject whenever the Type is not the same as the type of the element.

- The serialization of the DataObject is the same as if the Type for the DataObject had no inheritance at all, that is as if all the properties in Type.getProperties() were declared within the type.

XMLHelper Interface

```
public interface XMLHelper
{
   XMLDocument load(String inputString);
   XMLDocument load(InputStream inputStream) throws IOException;
   XMLDocument load(InputStream inputStream, String locationURI, Object
options) throws IOException;
   XMLDocument load(Reader inputReader, String locationURI, Object options)
throws IOException;
   void save(XMLDocument xmlDocument, OutputStream outputStream, Object
options) throws IOException;
   void save(XMLDocument xmlDocument, Writer outputWriter Object options)
throws IOException;
   void save(DataObject dataObject, String rootElementURI, String
rootElementName, OutputStream outputStream) throws IOException;
   String save(DataObject dataObject, String rootElementURI, String
rootElementName);
   XMLDocument createDocument(DataObject dataObject, String rootElementURI,
String rootElementName);

   XMLHelper INSTANCE = HelperProvider.getXMLHelper();
}
```

Creating DataObjects from XML

Using XMLHelper it is easy to convert between XML and DataObjects. The following example shows how to get a DataObject from XML:

```
String poXML =
  "<purchaseOrder orderDate='1999-10-20'>"+
  "  <shipTo country='US'>"+
  "    <name>Alice Smith</name>"+
  "    <street>123 Maple Street</street>"+
  "    <city>Mill Valley</city>"+
  "    <state>PA</state>"+
  "    <zip>90952</zip>"+
  "  </shipTo>"+
  "</purchaseOrder>";
```

```
DataObject po = XMLHelper.INSTANCE.load(poXML).getRootObject();
```

Creating DataObjects from XML documents

It is possible to convert to and from XML documents to build DataObject trees, which is useful when assembling DataObjects from several data sources. For example, suppose the global elements for shipTo and billTo were declared in the PurchaseOrder XSD:

```
<element name="shipTo" type="USAddress"/>
<element name="billTo" type="USAddress"/>
```

To create the shipTo DataObject from XML:

```
String shipToXML =
  "<shipTo country='US'>"+
  "  <name>Alice Smith</name>"+
  "  <street>123 Maple Street</street>"+
  "  <city>Mill Valley</city>"+
  "  <state>PA</state>"+
  "  <zip>90952</zip>"+
  "</shipTo>";
DataObject shipTo = XMLHelper.INSTANCE.load(shipToXML).getRootObject();
purchaseOrder.set("shipTo", shipTo);
```

To convert the billTo DataObject to XML:

```
String billToXML = XMLHelper.INSTANCE.save(billTo, null, "billTo");
System.out.println(billToXML);
```

This produces:

```
<?xml version="1.0" encoding="UTF-8"?>
<billTo country="US">
  <name>Robert Smith</name>
  <street>8 Oak Avenue</street>
  <city>Mill Valley</city>
  <zip>95819</zip>
</billTo>
```

Creating XML without an XSD

XMLHelper can be used without an XSD. In the TypeHelper Customer example, a Customer Type was defined dynamically without an XSD and then customer1 was created. To save customer1 to XML:

```
XMLHelper.INSTANCE.save(customer1, "http://example.com/customer", "customer",
stream);
```

This produces the following XML:

```
<?xml version="1.0" encoding="UTF-8"?>
<customer xsi:type="Customer" custNum="1" firstName="John" lastName="Adams"
  xmlns="http://example.com/customer"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"/>
```

An example of using the XML Helper in a simple web services client is found in the section Web Services Client using XMLHelper.

XMLDocument

An XMLDocument represents an XML Document containing a graph of DataObjects.

XMLHelper creates and serializes XMLDocument objects. An XMLDocument enables a program to access parts of an XML Document.

XMLDocuments do not change the state of any input DataObjects and ignore any containers.

XMLDocument Methods

- getRootObject() returns the root DataObject for the XMLDocument.
- getRootElementURI() returns the targetNamespace URI for the root element. If there is no targetNamespace URI, returns null.
- getRootElementName() returns the name of the root element.
- getEncoding() returns the encoding of the document, or null if not specified. The default value is "UTF-8". Specification of other values is implementation-dependent.
- setEncoding(String encoding) sets the XML encoding of the document, or null if the encoding is not specified.
- isXMLDeclaration() returns true if the document contains an XML declaration. The default value is true to enable new documents to contain the declaration.
- setXMLDeclaration(boolean xmlDeclaration) sets the XML declaration version of the document.
- getXMLVersion() returns the XML version of the document, or null if not specified. The default value is "1.0". Specification of other values is implementation-dependent.
- setXMLVersion(String xmlVersion) sets the XML version of the document, or null if not specified.

- getSchemaLocation() returns the value of the schemaLocation declaration for the http://www.w3.org/2001/XMLSchema-instance namespace in the root element, or null if not present.
- setSchemaLocation(String schemaLocation) sets the value of the schemaLocation declaration.
- getNoNamespaceSchemaLocation() returns the value of the noNamespaceSchemaLocation declaration for the http://www.w3.org/2001/XMLSchema-instance namespace in the root element, or null if not present.
- setNoNamespaceSchemaLocation(String schemaLocation) sets the value of the noNamespaceSchemaLocation declaration.

The root element is a global element of the XML Schema that has a type compatible with the DataObject.

Example XMLDocument

Using this XML Schema fragment:

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="purchaseOrder" type="PurchaseOrderType"/>
  <xsd:complexType name="PurchaseOrderType">
```

and the following example XMLDocument fragment:

```
<?xml version="1.0"?>
<purchaseOrder orderDate="1999-10-20">
```

After loading this XMLDocument:

- DataObjectPtr is an instance of Type PurchaseOrderType.
- RootElementURI is null because the XSD has no targetNamespace URI.
- RootElementName is purchaseOrder.
- Encoding is null because the document did not specify an encoding.
- XMLDeclaration is true because the document contained an XML declaration.
- XMLVersion is 1.0.
- SchemaLocation and noNamespaceSchemaLocation are null because they are not specified in the document.

XMLDocument Interface

```
public interface XMLDocument
{
  DataObject getRootObject();
  String getRootElementURI();
  String getRootElementName();
  String getEncoding();
  void setEncoding(String encoding);
  boolean isXMLDeclaration();
  void setXMLDeclaration(boolean xmlDeclaration);
  String getXMLVersion();
  void setXMLVersion(String xmlVersion);
  String getSchemaLocation();
  void setSchemaLocation(String schemaLocation);
  String getNoNamespaceSchemaLocation();
  void setNoNamespaceSchemaLocation(String schemaLocation);
}
```

XSDHelper

An XSDHelper provides additional information when a Type or Property is defined by an XML Schema (XSD). Also, an XSDHelper can define Types from XSDs.

If SDO Types and Properties were not originally defined by an XSD, or if the original XSD declaration information is not available, the helper methods will return null or false. IsXSD() can be used to tell if the XSDHelper has information about a Type.

The original name and namespace from an XML Schema can found using the getLocalName() and getNamespaceURI() methods.

It is possible to tell if a Property is serialized as an XML element or attribute with the isElement() and isAttribute() methods.

XML Schema global elements and attributes can be found using the getGlobalProperty() method. This is the most common way to build XML documents with open content, by finding a global property with the XSDHelper and then setting the property on an open content DataObject.

XSD Appinfo may be returned for a DataObject through the getAppinfo() methods. Appinfo is commonly used to specify information specific to a service in an XSD that may be valuable for configuring that service.

Default XSDHelper

The default XSDHelper is available from the INSTANCE field.

Generating XSDs

The XSDHelper can generate a new XSD for Types that do not already have an XSD definition. This is useful when the source of the Types come from services in another domain, such as relational databases, programming languages and UML. The generated XSD format is described later in the Generation of XSD section.

If an XML Schema was originally used to define the Types, that original XSD should be used instead of generating a new XSD. If a new XML Schema is generated when one already exists, the generated schema and the original schema will **not** be compatible and will validate different documents. The XMLHelper will follow the original XSD if one exists, otherwise it will follow a generated XSD.

XSDHelper Interface

The XSDHelper has methods to:

- Return the original XML local name for Types and Properties
- Return the namespace uri for a Property
- Identify if a Property is represented as an XML element or Attribute
- Identify if a Type allows XML mixed content

- Determine if a Type is defined from an XSD
- Return Properties for global elements and attributes
- Return the appinfo for Types and Properties
- Define new Types and Properties from XML Schemas
- Generate XML Schemas from Types and Properties

```
public interface XSDHelper
{
  String getLocalName(Type type);
  String getLocalName(Property property);
  String getNamespaceURI(Property property);

  boolean isAttribute(Property property);
  boolean isElement(Property property);
  boolean isMixed(Type type);
  boolean isXSD(Type type);

  Property getGlobalProperty(String uri, String propertyName, boolean
isElement);
  String getAppinfo(Type type, String source);
  String getAppinfo(Property property, String source);

  List /*Type*/ define(String xsd);
  List /*Type*/ define(Reader xsdReader, String schemaLocation);
  List /*Type*/ define(InputStream xsdInputStream, String schemaLocation);

  String generate(List /*Type*/ types);
  String generate(List /*Type*/ types, Map /*String String*/
namespaceToSchemaLocation);

  XSDHelper INSTANCE = getXSDHelper();
}
```

DataHelper

The DataHelper provides helper methods for working with DataObjects, and values used with DataObjects.

Methods are available for converting values between data types.

Default DataHelper

The default DataHelper is available from the INSTANCE field.

DataHelper Interface

```
public interface DataHelper
{
  Date toDate(String dateString);

  Calendar toCalendar(String dateString);
  Calendar toCalendar(String dateString, Locale locale);

  String toDateTime(Date date);
  String toDuration(Date date);
  String toTime(Date date);
  String toDay(Date date);
  String toMonth(Date date);
  String toMonthDay(Date date);
  String toYear(Date date);
  String toYearMonth(Date date);
  String toYearMonthDay(Date date);

  String toDateTime(Calendar calendar);
  String toDuration(Calendar calendar);
  String toTime(Calendar calendar);
  String toDay(Calendar calendar);
  String toMonth(Calendar calendar);
  String toMonthDay(Calendar calendar);
  String toYear(Calendar calendar);
  String toYearMonth(Calendar calendar);
  String toYearMonthDay(Calendar calendar);

  DataHelper INSTANCE = HelperProvider.getDataHelper();
}
```

HelperProvider

A HelperProvider obtains specific default helpers, and other implementation-specific objects, used by a Java implementation of SDO.

HelperProvider is an implementation class that must implement at least the following methods.

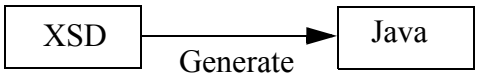
HelperProvider Class

```
package commonj.sdo.impl;

public class HelperProvider
{
  public static CopyHelper getCopyHelper();
```

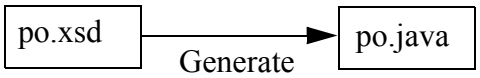
```
  public static DataFactory getDataFactory();
  public static DataHelper getDataHelper();
  public static EqualityHelper getEqualityHelper();
  public static TypeHelper getTypeHelper();
  public static XMLHelper getXMLHelper();
  public static XSDHelper getXSDHelper();
  public static Resolvable createResolvable();
  public static Resolvable createResolvable(Object target);
}
```

Generating Java from XML Schemas

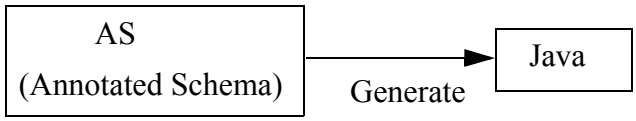
Generating Java from XML Schemas (XSD) is straightforward. An XML schema is the input to a code generator which produces Java files. This process applies to all methods that import or define Types and Properties from XML Schemas, such as XSDHelper.define().

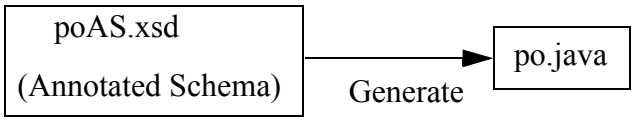


For example, to generate Java from the purchase order schema po.xsd, the process is:

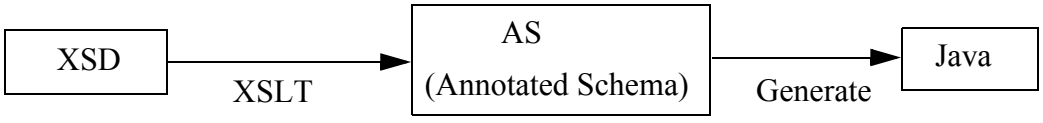


When customizing the default mapping, SDO annotations are added to the schema. This is called an Annotated Schema (AS). The AS is used to generate the Java. The annotated purchase order schema could be called poAS.xsd. The AS is important because all SDO implementations using the same AS would produce the same Java interfaces as defined in the Java code generation section.

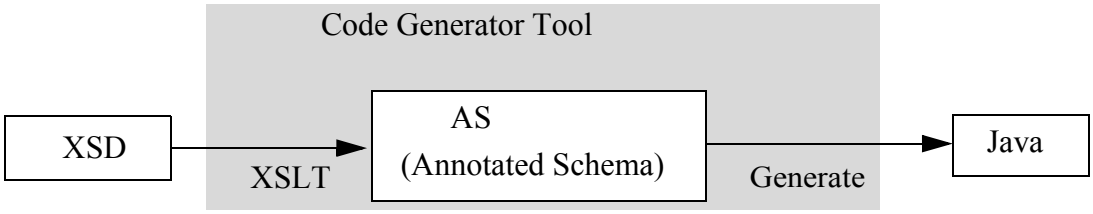




Often it is desirable for the original schema and the Annotated Schema to be different files with an automated process for producing the AS. XSLT is one way to automate the process where the annotations are stored in a side XSLT file and the annotations are applied by running XSLT. XSLT is an example of any process that annotates an XSD to produce an AS.

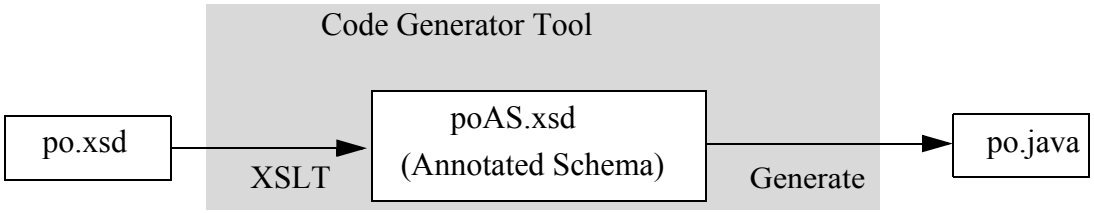


Frequently creation of annotations is done automatically by a code generation tool. In this case the XSLT and AS may be hidden within the implementation of the tool. This is very convenient in practice because the code generation tool can produce intelligent overrides and customizations in a product-specific fashion without creating any extra files or overhead.



Even though the AS may be hidden within a tool, every tool must provide a compliance option to produce the AS if requested. Also every tool must provide a compliance option to accept an AS without further modification as the input for code generation. This insures that an AS will produce the same Types, Properties, and generated interfaces for all implementations.

For the case of purchase order.xsd used with a code generator tool, the example is:



In addition to po.java, all other Java interfaces corresponding to declarations in poAS.xsd will be generated.

An example of an XSLT that will add the sdo:name annotation to an XSD for the purchase order XSD

```
<xsd:complexType name="PurchaseOrderType" sdo:name="purchaseOrder">
  <xsd:sequence>
    <xsd:element name="shipTo" type="USAddress"/>
    <xsd:element name="billTo" type="USAddress"/>
  </xsd:sequence>
</xsd:complexType>
```

is this XSLT template, which matches a complex type declaration of name PurchaseOrderType and copies the declaration while adding the sdo:name attribute.

```
<!-- Map the name of PurchaseOrderType to PurchaseOrder -->
<xsl:template match="xsd:complexType[@name='PurchaseOrderType']">
  <xsl:copy>
    <xsl:call-template name="copyAttributes" />
    <xsl:attribute name="sdo:name">PurchaseOrder</xsl:attribute>
    <xsl:apply-templates/>
```

```
  </xsl:copy>
</xsl:template>
```

XSD, SDO, and Java Names

In most cases, the names in XSD, SDO, and Java are identical.

When they are not identical, an annotated XSD declares the SDO names. The names in SDO and Java are identical when Java is used. The following are the naming rules.

1. All SDO Type names in a URI and all Property names in Type.getProperties() must be unique and non-null.
2. SDO does not specify name any mangling but enables and sometimes requires name overrides with sdo:name
3. If an XSD declaration would result in a duplicate name, sdo:name must be specified in the XSD file.
4. If an XSD type definition has no name, its name is the same as the next named enclosing declaration. If that is a duplicate then sdo:name must be used.
5. It is possible to use an automated **annotation utility** to produce an annotated XSD with the SDO annotations. The utility's output is the formally annotated XSD, producing a completely declarative description of the SDOs. Such a utility may implement any name mangling algorithm, allowing choice of mangling conventions. Tools may embed annotation utilities and must, at user option, make the formally annotated XSD available.
6. It is recommended that the name mangling algorithm ensure names use a character set compatible with common programming languages such as Java.

Management of annotated XSDs

Annotated XSDs can be hard to manage if the original XSD can be updated, or when it is convenient to have more than one set of SDO annotations used with the same XSD.

For example, one set of annotations can be useful for a server that will use generated code, while another set is more appropriate for clients that will not be using generated code.

The recommended approach, when directly annotating an XSD is not desirable, is to use XSLT as the annotation utility for XSD. XSLT is a widely supported standard for processing XML documents that is both simple and flexible. Any XSLT processor can combine the original XSD with the SDO annotations in an XSTL stylesheet to produce the formal annotated XSD used to define the SDO Types and Properties. Tools can automate the use of annotation utilities.

Java Interface Specification

Data Objects may be dynamic or static. When they are static Data Objects, an interface is generated following the pattern described in the tables below.

The implementation of the static interfaces must implement the DataObject interface, enabling all Data Objects, whether static or dynamic, to be used with the DataObject interface. When static interfaces are used, the types of associated Data Objects must also be static or dynamic subclasses of static interfaces to meet Java type requirements. The behavior of implementations of these interfaces must be identical whether called through the generated or DataObject interfaces.

The generation pattern described here is based on the Java Beans specification version 1.0.1, sections 8.3.1 and 8.3.2 http://java.sun.com/products/javabeans/docs/spec.html Because the generated interface does not depend on SDO, it is possible to use the same interfaces in any context - a client of the generated interfaces does not need to be aware of SDO or have SDO on the classpath to compile against the generated interfaces. Software already using the bean pattern may be able to upgrade to SDOs without change.

Each Type generates one interface. When [propertyName] and [typeName] appear, the first letter is capitalized. [javaType] is property.getType().getInstanceClass(). Each row specifying a method is generated when the expression for the property in the left column is true. The package defaults to "defaultPackage" or "noNamespace" and is overridden using sdoJava:package when generating from XML Schema. List is java.util.List. Boolean is the Java primitive boolean java.lang.Boolean.TYPE.

When a Type is generated, type.getInstanceClass() will return that java interface, and type.isInstance() will return the same results as type.getInstanceClass().isInstance(). Type.uri is unchanged by code generation. Generated Types may only be defined for Types where type.dataType is false. If [javaType] is one of the built in types in the DataObject interface, an implementation must have the same behavior as the corresponding method on the DataObject interface. For example, the generated method void setQuantity(long) behaves the same as setLong("quantity", long) and set("quantity", Long) on DataObject.

Compliance with generated interfaces is based on the ability to invoke methods specified by the generation pattern. It is valid to add any extra methods or extra inheritance useful to an implementation or based on additional metadata. It is also valid for the interface inheritance to be factored so that a required method is in an inherited interface. Both of these cases do not interfere with the ability to invoke the methods specified by the patterns. In particular, the interface may extend DataObject, and the implementation must always implement the DataObject interface.

Java code generation when the SDO source comes from an XSD uses the sdo and sdoJava annotations to determine the Java mapping. Because the names used are the same as in the XSD,

it is often important to annotate the XSD with sdo:name to produce valid Java code, as explained in the section on XSD, SDO, and Java names. In particular, sdo:name, sdoJava:instanceClass, and sdoJava:package annotations set the name, instance class, and package used when generating Java. All SDO Java generators using the same annotated XSD as input will produce the same Java interfaces when measured by invocation compliance above.

The sdo:javaPackage value will be used as the Java package name for generated classes. If "sdoJava:package" is not specified, a SDO-aware code generator tool will generate a new Java package name, virtually adding sdo:javaPackage annotation to the original XSD. Then, the tool will use the annotated schema to generate SDO. Such tool must be able to serialize the annotated schema at user request. If no sdo:javaPackage is present in the Annotated Schema, the Java interfaces are generated into the package "defaultPackage" or "noNamespace" respectively, based on whether a target namespace is declared in the XSD.

Java accessors with Types that have both an object and a primitive representation in Java (int and Integer for example) may be generated with either form and still be compliant. By allowing the code generator to choose between the primitive and object representations, the most useful and efficient representation may be selected. Users of these interfaces compiling with JDK 1.5 or later can write code independent of the choice of representation because of the autoboxing feature of the Java compiler.

Code generation template

Type	Java
For each Property in type.getProperties():	`public interface [typeName]` `{`
many = false && [javaType] != boolean	`  [javaType] get[propertyName]();`
many = false && [javaType] = boolean	`  [javaType] is[propertyName]();`
many = false && readOnly = false	`  void set[propertyName]([javaType]);`
many = true	`  List /*javaType*/ get[propertyName]();`

where

- [typeName] = type.name with the first character Character.toUpperCase().
- [propertyName] = property.name with the first character Character.toUpperCase().
- [javaType] = property.getType().getInstanceClass()
- List = java.util.List

It is permissible for code generated with J2SE 1.5 or later to generate many=true List methods of the form:

- `List<[javaType]> get[propertyName]();`

For convenience, code generators may at their discretion use the following pattern for a typed create method when a containment property's type is a DataObject type:

- `[javaType] create[propertyName]()`

This method is identical in behavior to DataObject.create([propertyName]).

For convenience, code generators may at their discretion use the following pattern for isSet/unset methods:

- `boolean isSet[propertyName]()`
- `void unset[propertyName]()`

These methods are identical in behavior to DataObject.isSet([propertyName]) and DataObject.unset([propertyName]).

These convenience options are not required to be offered by compliant SDO Java code generators. An implementation is required to provide an option that will generate SDO interfaces without content additional to SDO.

When generating code, it is possible for the accessor names to collide with names in the DataObject interface if the model has property names in the following set and their type differs from the return type in DataObject: changeSummary, container, containmentProperty, dataGraph, rootObject, sequence, or type.

Nested Java interfaces

When nested interfaces are supported by the code generator and enabled, interfaces for anonymous complex types are generated with a nesting that reflects their structure in an XML schema. Whether to nest is controlled when using XML Schema with the sdoJ:nestedInterfaces attribute. Nested interfaces are nested in the same interface that contains the accessors for the complex type's enclosing element. Nested interfaces have the same name whether nested or not. Since Type names are unique within a URI, all interface names in a package are unique also. Code that uses generated interfaces can be automatically converted to and from the nested style by using many development tools' "organize imports" function.

Notes: The nesting of interfaces does not necessarily affect the structure of implementation classes.

Example of generated interfaces

For the purchase order XSD without any annotations, the following are the minimal Java interfaces generated:

```
package noNamespace;

public interface PurchaseOrderType
{
  USAddress getShipTo();
  void setShipTo(USAddress value);
  USAddress getBillTo();
  void setBillTo(USAddress value);
  String getComment();
  void setComment(String value);
  Items getItems();
  void setItems(Items value);
  String getOrderDate();
  void setOrderDate(String value);
}

public interface USAddress
{
   String getName();
  void setName(String value);
  String getStreet();
  void setStreet(String value);
  String getCity();
  void setCity(String value);
  String getState();
  void setState(String value);
  BigDecimal getZip();
  void setZip(BigDecimal value);
  String getCountry();
  void setCountry(String value);
}
```

When interfaces are not nested (flat):

```
public interface Items
{
  List /*Item*/ getItem();
}

public interface Item
{
  String getProductName();
```

```
  void setProductName(String value);
  int getQuantity();
  void setQuantity(int value);
  BigDecimal getUSPrice();
  void setUSPrice(BigDecimal value);
  String getComment();
  void setComment(String value);
  String getShipDate();
  void setShipDate(String value);
  String getPartNum();
  void setPartNum(String value);
}
```

When interfaces are nested:

```
public interface Items
{
  List /*Item*/ getItem();

  interface Item
  {
    String getProductName();
    void setProductName(String value);
    int getQuantity();
    void setQuantity(int value);
    BigDecimal getUSPrice();
    void setUSPrice(BigDecimal value);
    String getComment();
    void setComment(String value);
    String getShipDate();
    void setShipDate(String value);
    String getPartNum();
    void setPartNum(String value);
  }
}
```

Java Serialization of DataObjects

To enable java.io.Serialization of DataObjects between different Java implementations, a format has been defined. This format contains all of the information in DataObjects but does not write anything that is tied to a specific Java implementation of SDO, into an ObjectOutputStream. This format is applicable when a DataGraph is not used. The java.io.Serialization for DataGraphs is in the section DataGraph XML Serialization.

The format supports one or many DataObjects from one or many trees of DataObjects, possibly intermixed with any other serializable Java Objects, in the same stream.

The format is made available by an implementation of a DataObject with the following writeReplace method implementation. The DataObject implementation does not need to use the java.io.Externalizable interface. The method may have any access modifier:

```
  Object writeReplace() throws ObjectStreamException
  {
    return new ExternalizableDelegator(this);
  }
```

The same ExternalizableDelegator class is used in every SDO implementation. It writes a common minimal class descriptor to the ObjectOutput stream, but all the behavior is delegated to the implementation through the HelperProvider.

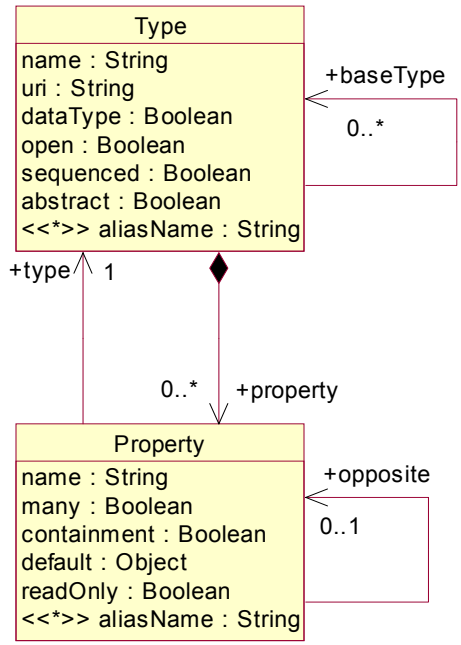
```
package commonj.sdo.impl;

/**
 * Delegates DataObject serialization while ensuring implementation
 * independent java.io.Serialization.  An implementation of DataObject
 * must return an ExternalizableDelegator from its writeReplace() method.
 *
 * The root DataObject is the object returned from do.getRootObject() where
 *   do is the DataObject being serialized in a java.io.ObjectOutputStream.
 *   When do.getContainer() == null then do is a root object.
 *
 * The byte format for each DataObject in the stream is:
 * [0] [path] [root]   // when do is not a root object
 * [1] [rootXML]       // when do is a root object
 *
 * where:
 * [0] is the byte 0, serialized using writeByte(0).
 * [1] is the byte 1, serialized using writeByte(1).
 *
 * [path] is an SDO path expression from the root DataObject to the serialized
 *          DataObject such that root.getDataObject(path) == do.
 *          Serialized using writeUTF(path).
 *
```

```
 * [root] is the root object serialized using writeObject(root).
 *
 * [rootXML] is the GZip of the XML serialization of the root DataObject.
 *          The XML serialization is the same as
 *         XMLHelper.INSTANCE.save(root, "commonj.sdo", "dataObject", stream);
 *          where stream is a GZIPOutputStream, length is the number of bytes
 *          in the stream, and bytes are the contents of the stream.
 *          Serialized using writeInt(length), write(bytes).
 *
 */
public class ExternalizableDelegator implements Externalizable
{
  public interface Resolvable extends Externalizable
  {
    Object readResolve() throws ObjectStreamException;
  }

  static final long serialVersionUID = 1;
  transient Resolvable delegate;

  public ExternalizableDelegator()
  {
    delegate = HelperProvider.createResolvable();
  }

  public ExternalizableDelegator(Object target)
  {
    delegate = HelperProvider.createResolvable(target);
  }

  public void writeExternal(ObjectOutput out) throws IOException
  {
    delegate.writeExternal(out);
  }

  public void readExternal(ObjectInput in) throws IOException,
                                          ClassNotFoundException
  {
    delegate.readExternal(in);
  }

  public Object readResolve() throws ObjectStreamException
  {
    return delegate.readResolve();
  }
}
```

SDO Model for Types and Properties

This SDO model describes SDO Types and Properties.

It contains the same information as is in the SDO interfaces shown in a model form, as a UML class diagram and as an XML Schema sdoModel.xsd.



Each of the Properties in the SDO model correspond to accessors on the Type and Property interfaces, as shown below in the tables. The model of Types and Properties is defined by the file sdoModel.xml.

Type and Property have open content so that additional new properties can be used even if they are not declared on the Type and Property interface. The Properties in AliasInfo, JavaInfo, and XSDInfo are set using the open content on Type and Property, as shown in the "applies to" column in the tables below.

Type Properties

Type has Properties: name, uri, dataType, open, sequenced, abstract, baseType, property, and aliasName.

Type model	Index	Type accessor

name	3	getName()
uri	4	getURI()
dataType	5	isDataType()
open	6	isOpen()
sequenced	7	isSequenced()
abstract	8	isAbstract()
baseType	0	getBaseTypes()
property	1	getDeclaredProperties()
aliasName	2	getAliasNames()

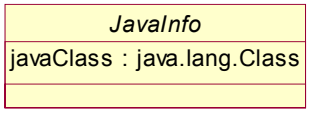
Property Properties

Property has Properties: name, many, containment, default, readOnly, type, opposite, and aliasName.

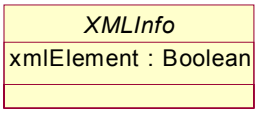
Property model	Index	Property accessor
name	1	getName()
many	2	isMany()
containment	3	isContainment()
default	4	getDefault()
readOnly	5	isReadOnly()
type	6	getType()
opposite	7	getOpposite()
aliasName	0	getAliasNames()

JavaInfo

JavaInfo is used when there is a mapping for the Java language. It is set on Types that are DataTypes and is the same as the name of the class returned from getInstanceClass() on the Type interface.



JavaInfo model	Type accessor	applies to
javaClass	getInstanceClass()	Type



XMLInfo is used when there is a mapping for XML. xmlElement is set on Properties that are represented as XML elements when the value is true. If no value is present there is no information about the mapping. If the value is false, it indicates that it is not an element, but it does not imply that there is mapping to an XML attribute. The URI of XMLInfo is "commonj.sdo/xml".

XMLInfo model	Property accessor	applies to
`xmlElement`	`get(xmlElement)`	`Property`

Standard SDO Types

These are the predefined SDO Types that are always available from either:

- TypeHelper.INSTANCE.getType("commonj.sdo", String typeName).
- DataGraph.getType("commonj.sdo", String typeName).

SDO Data Types

The term *SDO data type* refers to an SDO Type where isDataType() = true. None of the types have any Properties unless noted. All values are false unless noted.

The Java instance class is the expected type of the instance returned through the DataObject.get(property) method. Other DataObject methods of the form getXXX(property) where XXX is another type such as int or String are conversions between the get(property) value and the XXX type as shown in the SDO type conversion table. The same is true for the setXXX(property, value) methods and the set(property, value) method. When code is generated with accessors of type XXX, the behavior is identical to the getXXX(property) and setXXX(property) methods.

The SDO Types are applicable across all languages mapped into SDO. The SDO Java Types are additional types specifically used in Java representing the object wrappers for primitive Java types. When an SDO Type is used in a mapping from another technology to SDO, implementations in Java specify one of the corresponding Java types. For example, the predefined XSD int SimpleType maps to the SDO Type of Int. When SDO is used in Java, an implementation may select either the SDO Int Type, or the SDO Java IntObject Type as the actual type used to represent the XSD int. When crossing between languages, the DataType mapping is between the SDO Types in each language.

SDO Type URI = commonj.sdo	Java instance Class
`Boolean`	`boolean`
`Byte`	`byte`
`Bytes`	`byte[]`
`Character`	`char`
`Date`	`java.util.Date`
`DateTime`	`String`
`Day`	`String`
`Decimal`	`java.math.BigDecimal`
`Double`	`double`
`Duration`	`String`
`Float`	`float`
`Int`	`int`

Integer	java.math.BigInteger
Long	long
Month	String
MonthDay	String
Object	java.lang.Object
Short	short
String	String
Strings	List<String>
Time	String
URI	String
Year	String
YearMonth	String
YearMonthDay	String

Each DataType has a String representation and may be converted to and from the String representation to its instance class, if that instance class is different from String. Numeric DataTypes have a precision in terms of a number of bits. For example, 32 bits signed indicates 1 sign bit and 31 value bits, with a range of -2^31 to 2^31-1. The String representation of DateTime, Duration, Time, Day, Month, MonthDay, Year, YearMonth, and YearMonthDay follows the lexical representation defined in XML Schema for the corresponding data types: dateTime, duration, time, gDay, gMonth, gMonthDay, gYear, gYearMonth, and Date respectively.

List<String> represents a List where all the values are of type String. On JDKs earlier than 1.5, this is the List interface where all values are Strings. List<String> are converted to a String by inserting a space character between each value. String is converted to List<String> with contents as defined by the String.split("\\s") method in the JDK, which splits the string on whitespace boundaries.

SDO Type	Precision	String Representation
Boolean	1 bit	'true' | 'false'
Byte	8 bits unsigned	[0-9]+
Bytes		[0-9A-F]+
Character		any character
Date		'-'?yy'-'mm'-'dd'T'hh':'mm':'ss('.'s+)? 'Z'?
DateTime		'-'?yy'-'mm'-'dd'T'hh':'mm':'ss('.'s+)? zz?
Day		'---'dd zz?
Decimal		('+'|'-')? [0-9]* ('.'[0-9]*)? (('E'|'e') ('+'|'-')? [0-9]+)?
Duration		'-'?'P'(yy'Y')? (mm'M')? (dd'D')? ('T'(hh'H')? (mm'M')? (ss'S'('.'s+)?)?)?
Double	IEEE-754 64 bits	Decimal | 'NaN' | '-NaN' | 'Infinity' | '-Infinity'
Float	IEEE-754 32 bits	Decimal | 'NaN' | '-NaN' | 'Infinity' | '-Infinity'

Int	32 bits signed	('+'|'-')? [0-9]+
Integer		('+'|'-')? [0-9]+
Long	64 bits signed	('+'|'-')? [0-9]+
Month		'--'mm zz?
MonthDay		'--'mm'-'dd zz?
Short	16 bits signed	('+'|'-')? [0-9]+
String		any characters
Strings		any characters separated by whitespace
Time		hh':'mm':'ss('.'s+)? zz?
URI		any characters
Year		'-'?yy zz?
YearMonth		'-'?yy'-'mm zz?
YearMonthDay		'-'?yy'-'mm'-'dd zz?

where

- [0-9] any digit, [0-9A-F] any hexidecimal digit.
- '-' single quotes around a literal character, () for higher precedence, | for choice.
- ? occurs zero or one time, * occurs zero or more times, + occurs one or more times.
- Decimal lexical representation is valid for Double and Float.
- yy year, mm month, dd day, hh hour, mm minute, ss second, s fractional second
- zz time zone (('+'|'-')hh':'mm)|'Z' where hh time zone hour, mm time zone minute.
- Date will accept the same lexical format as DateTime but will normalize to the Z time zone.

SDO Java Type URI = commonj.sdo/java	Java instance Class
BooleanObject	java.lang.Boolean
ByteObject	java.lang.Byte
CharacterObject	java.lang.Character
DoubleObject	java.lang.Double
FloatObject	java.lang.Float
IntObject	java.lang.Integer
LongObject	java.lang.Long
ShortObject	java.lang.Short

If a value is null and a conversion to (byte, char, double, float, int, long, short) is requested by a DataObject.getXXX() method, 0 is returned. If a value is null and a conversion to boolean is requested by a DataObject.getBoolean() method, false is returned. This also applies to generated accessors.

DataObject.getDate() and setDate() on a property of type String

java.util.Date values for String properties are converted using the DateTime DataType String representation for the Z time zone, for example 1999-05-31T13:20:00Z. The output is always in the Z time zone. Null Date will return a null String. DataHelper toDate(String) and toString(Date) perform this conversion.

The output using this example java.text.SimpleDateFormat is compliant:

```
DateFormat f = new SimpleDateFormat("yyyy-MM-dd'T'HH:mm:ss'.'SSS'Z'");
f.setTimeZone(TimeZone.getTimeZone("GMT"));
String dateString = f.format(date);
```

DataObject.getString() and setString() on a property of type Date

String values for java.util.Date properties are converted using the DateTime DataType String representation. The String may be right truncated, where any omitted field is assumed to be 0. If the time zone is not specified, Z is assumed. Null String will return a null Date. If precision beyond milliseconds is specified it may not be preserved since java.util.Date precision is milliseconds. An implementation may accept a wider range of Strings for conversion to Date, for example RFC 822 time zones are supported by SimpleDateFormat. DataHelper toDate(String) and toString(Date) perform this conversion.

The following example use of java.text.SimpleDateFormat is compliant for converting String to Date when all fields are present:

```
DateFormat f = new SimpleDateFormat("yyyy-MM-dd'T'HH:mm:ss'.'SSSz");
Date date = f.parse(dateString.replaceFirst("([+\\-]..):", "$1"));
```

Conversion between java.util.Date, Calendar and date-times

These conversions are performed by methods on the DataHelper. When creating a Calendar from the date-times (values of type DateTime, Day, Duration, Month, MonthDay, Year, YearMonth, YearMonthDay) the calendar is created with the current time and date. Each field present in the date-time value is set on the calendar, leaving fields not present in the calendar unchanged. The time zone offset is set to (time zone hours*60+time zone minutes)*1000.

When creating a date-time from a Calendar, each value allowed in the format is taken from the corresponding field in the Calendar. The time zone hours is the calendar's time zone offset / 1000 / 60 and the minutes to the time zone offset / 1000 % 60.

Calendars are converted to and from Dates through the Calendar's getTime() and setTime() methods. Dates and date-times conversions are defined to produce the same result as conversion through Calendar as an intermediate step.

DataObject.getString() and setString() on a property of type Bytes

Bytes are converted to String by converting each byte into the hexadecimal two-digit equivalent using the characters [0-9A-F]. The 0 index of the byte array becomes the 0th and 1st index of the String, with subsequent values in order to the right. Null Bytes become null Strings. This representation is compatible with XML Schema hexBinary dataType canonical lexical representation. An example conversion of byte[] = { 10, 100 } becomes the String "0A64".

DataObject.getBytes() and setBytes() on a property of type String

Strings are converted to Bytes by converting each pair of characters from the hexadecimal two-digit equivalent using the characters [0-9A-Fa-f]. The 0th and 1st index of the String becomes the 0 index of the byte array, with subsequent values in order to the right. Null Strings become null Bytes. This representation is compatible with XML Schema hexBinary dataType lexical representation. An example conversion of the String "0A64" becomes byte[] = { 10, 100 }.

SDO Abstract Types

The following types may not be instantiated. They describe metadata for DataObjects, Types, and Properties. Types that may not be instantiated throw IllegalArgumentException from all create() methods.

SDO Abstract Type URI = commonj.sdo	Java instance Class	XSD Type

`ChangeSummaryType`
`  abstract=true`
`  dataType=true`
 	
ChangeSummaries are instances.
	`commonj.sdo.`
	`ChangeSummary`
		`ChangeSummaryType`
		`in the SDO`
		`namespace.`

DataObject abstract=true DataObjects are instances.	commonj.sdo. DataObject	Not applicable

Object
  abstract=true
  dataType=true
  isInstance()= true.
	java.lang.Object
	Values must support
	  toString() for
	  String value
		Not applicable

TextType
  abstract=true
  Property name="text"
  type="String" many="true"
 	
XML text uses this property.
	null	Not applicable

SDO Model Types

Type and Property describe themselves. The definition is:

SDO Model Types

Type
  name="Type"
  open=true
  uri="commonj.sdo"
 	
  Property name="baseType" many=true type="Type"
  Property name="property" containment=true many=true
  Property name="aliasName" many=true type="String"
  Property name="name" type="String"
  Property name="uri" type="String"
  Property name="dataType" type="Boolean"
  Property name="open" type="Boolean"
  Property name="sequenced" type="Boolean"
  Property name="abstract" type="Boolean"
type="Property"

```
Property
  name="Property"
  open=true
  uri="commonj.sdo"

  Property name="aliasName" many=true type="String"
  Property name="name" type="String"
  Property name="many" type="Boolean"
  Property name="containment" type="Boolean"
  Property name="type" type="Type"
  Property name="default" type="Object"
  Property name="readOnly" type="Boolean"
  Property name="opposite" type="Property"
```

SDO Type and Property constraints

There are several restrictions on SDO Types and Properties. These restrictions ensure Types and Properties for DataObjects are consistent with their API behavior. Behavior of ChangeSummaryType Properties is defined.

- Instances of Types with dataType=false must implement the DataObject interface and have isInstance(DataObject) return true.
- If a Type's instance Class is not null, isInstance(DataObject) can only be true when instanceClass.isInstance(DataObject) is true.
- Values of bidirectional Properties with type.dataType=false and many=true must be unique objects within the same list.
- Values of Properties with type.dataType=false and many=true cannot contain null.
- Property.containment has no effect unless type.dataType=false
- Property.default!=null requires type.dataType=true and many=false
- Types with dataType=true cannot contain properties, and must have open and sequenced=false.
- Type.dataType and sequenced must have the same value as their base Types' dataType and sequenced.
- Type.open may only be false when the base Types' open are also false.
- Instance classes in Java must mirror the extension relationship of the base Types.
- Properties that are bi-directional require type.dataType=false
- Properties that are bi-directional require that no more than one end has containment=true
- Properties that are bi-directional require that both ends have the same value for readOnly
- Properties that are bi-directional with containment require that the non-contaiment Property has many=false.
- Names and aliasNames must all be unique within Type.getProperties()

ChangeSummaryType Properties:

- Types may contain one property with type ChangeSummaryType.
- A property with type ChangeSummaryType must have many=false and readOnly=true.
- The scope of ChangeSummaries may never overlap. The scope of a ChangeSummary for a DataGraph is all the DataObjects in the DataGraph. If a DataObject has a property of type ChangeSummary, the scope of its ChangeSummary is that DataObject and all contained DataObjects. If a DataObject has a property of type ChangeSummay, it cannot contain any other DataObjects that have a property of type ChangeSummay and it cannot be within a DataGraph.
- ChangeSummaries collect changes for only the DataObjects within their scope.
- The scope is the same whether logging is on or off.
- Serialization of a DataObjects with a property of type ChangeSummaryType follows the normal rules for serializing a ChangeSummary.

Example Use of ChangeSummaryType

A common use of defining DataObject Types with a ChangeSummary is when wrapping specific existing types such as PurchaseOrders along with a ChangeSummary tracking their changes. A message header defined by the following XSD is an example.

```
<element name="message" type="PurchaseOrderMessageType"/>
<complexType name="PurchaseOrderMessageType">
  <sequence>
    <element name="purchaseOrder" type="po:PurchaseOrderType"/>
    <element name="changeSummary" type="sdo:ChangeSummaryType"/>
  </sequence>
</complexType>
```

The following is an example message document:

```
<message>
  <purchaseOrder orderDate="1999-10-20">
    <shipTo country="US">
        <name>Alice Smith</name>
    </shipTo>
    <comment>Hurry, my lawn is going wild!</comment>
  </purchaseOrder>
  <changeSummary>
    <USAddress sdo:ref="sdo:/purchaseOrder.0/shipTo" name="John Public"/>
  </changeSummary>
</message>
```

XML Schema to SDO Mapping

XML Schema declarations (XSD) are mapped to SDO Types and Properties following the principles outlined below. [2] [7] (The abbreviation XSD is used for both the XML Schema infoset and the XML Schema declarations used to build the infoset.)

This simple yet flexible mapping allows SDO DataObjects to represent XML documents following an XSD. The vast majority of XSD capabilities are mapped and several corner cases are included. XML documents without XSDs are also handled.

Sequenced DataObjects preserve detailed information about the order of XML elements.

This document describes the Mapping Principles, Mapping of XSD Types, Sequenced DataObject, Mapping of XSD elements and Attributes, Mapping of data types and XML document mapping. It also provides Examples.

Mapping Principles

Creating SDO Types and Properties from XML Schema is increasingly important as a great deal of structured information is described by XSDs. The following tables describe the mapping.

XML Schema Concept	SDO Concept	Java Concept
Schema	URI for Types	Package
Simple Type	Type, dataType=true SDO data types	Primitives, String, BigDecimal, etc.
Complex Type	Type, dataType=false SDO DataObjects	Interface
Attribute	Property within enclosing Type	getX(), setX() accessors
Element	Property within enclosing Type	getX(), setX() accessors

The general principles are that:

1. A Schema target namespace describes the URI for a set of Types.
2. SimpleType declarations describe data types, Types where isDataType() is true.
3. ComplexType declarations describe DataObjects, Types where isDataType() is false.
4. Within each ComplexType, the elements and attributes describe Properties in the corresponding enclosing Type.
5. Model groups (all, choice, sequence, group reference) are expanded in place and do not describe Types or Properties. There is no SDO or Java construct corresponding to groups in this specification.
6. Open content and mixed content map to Type.open.

7. Mixed content maps to Type.sequenced and uses the Text property for mixed text.
8. Order of element content maps to Type.sequenced.
9. XSD any and anyAttribute (wildcard) declarations are not required to map to Types or Properties.
10. We do not allow design changes that complicate the simple cases to solve more advanced cases.
11. The mapping input is an annotated XSD using the SDO annotations. The mapping output is SDO Types and Properties.
12. Normally, SDO names are the same as the XSD names. To change the SDO name user can annotate an XSD with sdo:name annotations. In some cases, for example in the case of duplicate component names in XSD, the original XSD names cannot be preserved in SDO. In such cases, an SDO-aware code generator tool will generate new names and virtually add sdo:name annotations to the original XSD. Then, the tool will use the Annotated Schema to generate SDO. Such tool must be able to serialize the Annotated Schema at user request.
13. This mapping specifies a minimum. Implementations may expand this mapping to perform additional functions as long as the mapping stated here works for all client code.

Mapping of XSD to SDO Types and Properties

There are a number of customizations that can be used to improve the mapping to SDO.

These are expressed as attributes in the SDO namespace for XML, "commonj.sdo/xml". The following XSD attributes in the SDO XML namespace are used to modify the constructed SDO model:

1. **name** - sets the SDO name to the name specified here. Applies to Type and Property. Used in ComplexType, SimpleType, element, and attribute declarations. The XSD type of the annotation is string.
2. **propertyType** - sets the Property's Type as specified by the QName value. Applies to Property. Used in element and attribute declarations. Property.type.dataType must be false. The XSD type must be IDREF, IDREFS, or anyURI, or restrictions of these types. The XSD type of the annotation is QName.
3. **oppositeProperty** - sets the Property opposite to be the property with the given name within the Type specified by **propertyType**. Applies to Property, making the property bidirectional. Used in element and attribute declarations. Property.type.dataType must be false. The XSD type must be IDREF, IDREFS, or anyURI or restrictions of these types. Requires sdo:propertyType on the property. Automatically creates the opposite property if one or both ends are specified in the XSD, with opposites bidirectional. The XSD type of the annotation is string.
4. **sequence**="true" - sets Type.sequenced to true. Applies to Type. Used in ComplexType declarations. A Sequenced Type has a Sequence for all XML Elements. The default is

false. If schema extension is used, the base complexType must also be marked sequence="true". The XSD type of the annotation is boolean.

5. **string="true" -** sets the SDO Type to String for XSD SimpleTypes as a means to override the instance class when the exact values must be preserved. Applies to Property. Used in element and attribute declarations. Same as sdo:dataType="sdo:String". The XSD type of the annotation is boolean.
6. **dataType** - sets the Property's type as specified by the QName value as a means to override the declared type. Applies to XML attributes and elements with simple content. Used in element and attribute declarations. The XSD type of the annotation is QName.
7. **aliasName** - add alias names to the SDO Type or Property. The format is a list of names separated by whitespace, each becoming an aliasName. Applies to Type and Property. The XSD type of the annotation is string.
8. **readOnly** - indicate the value of Property.readOnly. The format is boolean with default false. Applies to Property. Used in element and attribute declarations. The XSD type of the annotation is boolean.

Some customizations can be used to improve the mapping to the Java representation of SDO. This is expressed as attributes in the SDO namespace for Java, "commonj.sdo/java". The following XSD attributes in the SDO Java namespace are used to modify the constructed SDO model:

1. **package** - sets the Java package name to be the full Java package name specified. Enables Java code generators to determine the package for generated interfaces. Applies to the Schema element. If the Java package is not specified, Java code generators will place interfaces in the Java package named "defaultPackage" or "noNamespace". The XSD type of the annotation is string.
2. **instanceClass** - sets the Java instanceClass for the Type to be the full class name specified. Enables custom classes that may implement behavior appropriate to a specific type. Applies to SimpleTypes. The instance class must have a public String constructor for creating values, and the toString() method will be used to convert the instances to their XML representation. The instance class specified does not extend the base instance instance class in Java. An SDO Type with this specification does not have base Types. The XSD type of the annotation is string.
3. **extendedInstanceClass** - same as instanceClass except that the instance class specified must extend the base Type's instance class in Java. The SDO base Type relationship follows the schema base type relationship. The XSD type of the annotation is string.
4. **nestedInterfaces** - sets the nesting of anonymous complex types when generating Java code. Applies to the Schema element. When absent, the implementation may select either nested or non-nested interface generation. When present and true, nested interfaces are generated. When present and false, non-nested interfaces are generated. An implementation is only required to support generation of one style. The annotation has no effect on the name of Types or interfaces, which are the same whether nested or not, and

unique within a URI or package. Included schemas must have the same value (true, false, or absent) as the including schema. Imported schemas may have different values. The XSD type of this annotation is boolean.

In all tables, SDO Type and Property values that are not shown default to false or null, as appropriate. [URI] is the targetNamespace. Use sdo:name to override the names as desired.

XML Schemas

XML Schemas	SDO Package
Schema with targetNamespace `<schema targetNamespace=[URI]>`	[URI] is type.uri) for the types defined by this Schema.
Schema without targetNamespace `<schema>`	[URI] is null. Null is type.uri for the types defined by this Schema.

Schema with sdoJava:package
 	
`<schema`
`sdoJava:package="[PACKAGE]">`
	Java interfaces will be generated in Java package [PACKAGE].

XML Simple Types

XML simple types map directly to SDO types.

The mapping of XML Schema built-in simple types is defined in another section below. The Java instance class is the class for the values returned by DataObject.get(property). The notation [BASE].instanceClass indicates the instance class of the SDO Type corresponding to [BASE]. When deriving Simple Types by restriction, the base for the SDO Type follows the XSD SimpleType restriction base, unless an sdoJava:instanceClass is declared, which causes there to be no base relationship.

The value of the JavaInfo:instanceClass property for the SDO Type is set to the value in the Java Instance Class column.

When the XSD type is integer, positveInteger, negativeInteger, nonPositiveInteger, nonNegativeInteger, long, or unsignedLong, and there are facets (minInclusive, maxInclusive,

minExclusive, maxExclusive, enumeration) constraining the range to be within the range of int, then the Java instance class is int and the base is null unless the base Type's instance class is also int.

XML Simple Types	SDO Type	Java Instance Class

Simple Type with name
 	
`<simpleType name=[NAME]>`
`  <restriction base=[BASE]/>`
`</simpleType>`
	`Type name=[NAME]`
	`  base=[BASE]`
	`  dataType=true`
	`  uri=[URI]`
		`[BASE].instanceClass`

Simple Type Anonymous
 	
`<... name=[NAME] ...>`
`<simpleType>`
`  <restriction base=[BASE]/>`
`</simpleType>`
`</...>`
 	
[NAME]=enclosing element or attribute name
	`Type name=[NAME]`
	`  base=[BASE]`
	`  dataType=true`
	`  uri=[URI]`
	
	• [NAME] of the anonymous type is the same as the name of the enclosing element or attribute declaration.
		`[BASE].instanceClass`

Simple Type with sdo:name
 	
`<simpleType name=[NAME]`
`  sdo:name=[SDO_NAME]>`
`  <restriction base=[BASE]/>`
`</simpleType>`
	`Type name=[SDO_NAME]`
	`  base=[BASE]`
	`  dataType=true`
	`  uri=[URI]`
		`[BASE].instanceClass`

`Simple Type with abstract`
 	
`<simpleType name=[NAME]`
`  abstract="true">`
`  <restriction base=[BASE]/>`
`</simpleType>`
	`Type`
	`  abstract=true`
	`  base=[BASE]`
	`  dataType=true`
	`  uri=[URI]`
		`[BASE].instanceClass`

Simple Type with sdoJava:instanceClass
 	
`<simpleType name=[NAME]`
`  sdoJava:instanceClass =`
`  [INSTANCE_CLASS]>`
`</simpleType>`
	`Type name=[NAME]`
	`  dataType=true`
	`  uri=[URI]`
	
	btNo base Type
		`[INSTANCE_CLASS]`

Simple Type with sdoJava:extendedInstanceClass
 	
`<simpleType name=[NAME]`
 	
`sdoJava:extendedInstanceClass =`
`[INSTANCE_CLASS]>`
`<restriction base=[BASE]/>`
`</simpleType>`
	`Type name=[NAME]`
	`base=[BASE]`
	`dataType=true`
	`uri=[URI]`
		`[INSTANCE_CLASS]`

Simple Type with list of itemTypes
 	
`<simpleType name=[NAME]>`
`<list itemType=[BASE] />`
`</simpleType>`
	`Type name=[NAME]`
	`dataType=true`
	`uri=[URI]`
		`java.util.List`
		Entries in the List are of type
		`[BASE].instanceClass`

Simple Type with union
 	
`<simpleType name=[NAME]>`
`<union memberTypes=[TYPES]/>`
`</simpleType>`
	`Type name=[NAME]`
	`dataType=true`
	`uri=[URI]`
		`[TYPE].instanceClass`
		if all member types have the same SDO instanceClass where [TYPE] is the first SDO Type from [TYPES].
		
		`java.lang.Object`
		otherwise

XML Complex Types

XML Complex Types	SDO Type	Java Instance Class

Complex Type with empty content
 	
`<complexType name=[NAME] />`
	`Type name=[NAME]`
	`uri=[URI]`
	
	No Properties.
		`interface [NAME]`

Complex Type with content
 	
`<complexType name=[NAME] />`
	`Type name=[NAME]`
	`uri=[URI]`
	
	Properties for each element and attribute.
		`interface [NAME]`
		
		get/set pairs for each property following the Java Beans property pattern.

Complex Type Anonymous
 	
`<... name=[NAME] ...>`
`  <complexType  />`
`</...>`
 	
[NAME]=enclosing element name
	`Type name=[NAME]`
	`  uri=[URI]`
	
	• [NAME] of the anonymous type is the same as the name of the enclosing element declaration
		`interface [NAME]`

Complex Type with sdo:name
 	
`<complexType name=[NAME]`
`  sdo:name=[SDO_NAME] />`
	`Type name=[SDO_NAME]`
	`  uri=[URI]`
		`interface [SDO_NAME]`

Complex Type with abstract
 	
`<complexType name=[NAME}`
`  abstract="true">`
	`Type name=[NAME]`
	`  abstract=true`
	`  uri=[URI]`
		`interface [NAME]`

Complex Type with sdo:aliasName
 	
`<complexType name=[NAME]`
 	
`sdo:aliasName=[ALIAS_NAME]`
`/>`
	`Type name=[NAME]`
	`  aliasName=[ALIAS_NAME]`
	`  uri=[URI]`
		`interface [NAME]`

Complex Type extending a Complex Type
 	
`<complexType name=[NAME]>`
`  <complexContent>`
`  <extension base=[BASE]/>`
`  </complexContent>`
`</complexType>`
 	
or
 	
`<complexType name=[NAME]>`
`  <simpleContent>`
`  <extension base=[BASE]/>`
`  </simpleContent>`
`</complexType>`
	`Type name=[NAME]`
	`  base=[BASE]`
	`  uri=[URI]`
	
	`properties+=[BASE].properties`
	
	• Type.getProperties() maintains the order of [BASE].getProperties() and appends the Properties defined here.
		`interface [NAME]`
		`  extends [BASE]`

Complex Type with complex content restricting a Complex Type
 	
`<complexType name=[NAME]>`
`  <complexContent>`
`  <restriction base=[BASE]/>`
`  </complexContent>`
`</complexType>`
	`Type name=[NAME]`
	
	`properties=[BASE].properties`
	`  base=[BASE]`
	`  uri=[URI]`
	
	• Type.getProperties() maintains the order of [BASE].getProperties() and appends the Properties defined here.
	• When element and attribute declarations are in both the base type and the restricted type, no additional Properties are created and declarations inside the complex type are ignored.
	• When new element or attribute declarations are added in the restricted type that are not in the base type and restrict wildcard <any> and <anyAttribute> in the base, the element and attribute declarations are added as new Properties.
		`interface [NAME]`
		`  extends [BASE]`

Complex Type with simple content restricting a Complex Type
 	
`<complexType name=[NAME]>`
`  <simpleContent>`
`  <restriction base=[BASE]/>`
`  </simpleContent>`
`</complexType>`
	`Type name=[NAME]`
	`  base=[BASE]`
	`  uri=[URI]`
	
	`properties+=[BASE].properties`
	
	• Type.getProperties() maintains the order of [BASE].getProperties() and appends the Properties defined here.
		`interface [NAME]`
		`  extends [BASE]`

Complex Type with mixed content
 	
`<complexType name=[NAME]`
`  mixed="true" />`
	`Type name=[NAME]`
	`  open=true`
	`  sequenced=true`
	`  uri=[URI]`
	
	• The SDO Text Property is used for mixed text as an instance property.
	• Use getInstanceProperties() for reflection.
	• Sequence is typically used to access the values. DataObject and generated accessors also may be used.
		`interface [NAME]`

Complex Type with sdo:sequence
 	
`<complexType name=[NAME]`
`  sdo:sequence="true" />`
	`Type name=[NAME]`
	`  sequenced=true`
	`  uri=[URI]`
		`interface [NAME]`

Complex Type extending a SimpleType
 	
`<complexType name=[NAME]>`
`  <simpleContent>`
`  <extension base=[BASE]/>`
`  </simpleContent>`
`</complexType>`
	`Type name=[NAME]`
	`  uri=[URI]`
	`  Property:`
	`  name="value" type=[BASE]`
	
	• Properties are created for attribute declarations.
		`interface [NAME]`
		`{`
		`  [BASE] getValue();`
		`  void setValue([BASE]);`
		`}`
		Where `[BASE]` represents the instanceClass of the simpleType for the simple content.

Complex Type with open content
 	
`<complexType name=[NAME]>`
`  ...`
`  <any />`
`  ...`
`</complexType>`
	`Type name=[NAME]`
	`  open=true`
	`  uri=[URI]]`
	
	• No property required for <any>.
	• Use getInstanceProperties() for reflection.
	• DataObject and generated accessors also may be used to access the value.
	• If maxOccurs > 1, sequenced=true.
		`interface [NAME]`

Complex Type with open attributes
 	
`<complexType name=[NAME]>`
`  ...`
`  <anyAttribute />`
`  ...`
`</complexType>`
	`Type name=[NAME]`
	`  open=true`
	`  uri=[URI]`
	
	• No property required for <anyAttribute>.
	• Use getInstanceProperties() for reflection.
	• DataObject and generated accessors also may be used to access the value.
		`interface [NAME]`

Mapping of XSD Attributes and Elements to SDO Properties

Each XSD element or attribute maps to an SDO property.

The Property.containingType is the SDO Type for the enclosing ComplexType declaration.

The order of Properties in Type.getDeclaredProperties() is the lexical order of declarations in the XML Schema ComplexType. When extension is used, the Properties of the base type occur first in the Properties list.

If elements and attributes within a complexType, and its base types, have the same local name then unique names must be assigned by sdo:name. This ensures that all property names in Type.getProperties() are unique. Multiple elements with the same name and URI are combined into a single Property and the Type is sequenced, as described in the Mapping of XSD Elements section.

When creating a Property where the default or fixed value is not defined by the XSD, the Property's default is assigned based on its Type's instance class, property.getType().getInstanceClass() :

- Boolean has default false.
- Primitive numerics (Byte, Char, Double, Float, Int, Short, Long) have default is 0.
- Otherwise, the default is null.

Note that XSD anyType is a ComplexType and XSD anySimpleType is a SimpleType. They follow the normal mapping rules.

Mapping of XSD Attributes

XML Attribute	SDO Property

Attribute
 	
`<attribute name=[NAME]`
`type=[TYPE] />`
	`Property name=[NAME]`
	`type=[TYPE]`
	• DataObject accessors may enforce simple type constraints.

Attribute with sdo:name
 	
`<attribute name=[NAME]`
`sdo:name=[SDO_NAME]`
`type=[TYPE] />`
	`Property name=[SDO_NAME]`
	`type=[TYPE]`

Attribute with sdo:aliasName
 	
`<attribute name=[NAME]`
`sdo:aliasName=[ALIAS_NAME]`
`type=[TYPE] />`
	`Property name=[NAME]`
	`aliasName=[ALIAS_NAME]`
	`type=[TYPE]`

Attribute with default value
 	
`<attribute name=[NAME]`
`type=[TYPE]`
`default=[DEFAULT] />`
	`Property name=[NAME]`
	`type=[TYPE]`
	`default=[DEFAULT]`

Attribute with fixed value
 	
`<attribute name=[NAME]`
`type=[TYPE]`
`fixed=[FIXED] />`
	`Property name=[NAME]`
	`type=[TYPE]`
	`default=[FIXED]`

Attribute reference
 	
`<attribute ref=[ATTRIBUTE] />`
	`Property name=[ATTRIBUTE].[NAME]`
	`type=[ATTRIBUTE].[TYPE]`
	`default=[ATTRIBUTE].[DEFAULT]`
	• Use the XSDHelper to determine the URI of the attribute if the referenced attribute is in another namespace.

Attribute with sdo:string
 	
`<attribute name=[NAME]`
`type=[TYPE]`
`sdo:string="true" />`
	`Property name=[NAME]`
	`type=String`
	• The type of the property is SDO String
	• Used when the instance class for TYPE is not appropriate.

Attribute referencing a DataObject with sdo:propertyType
 	
`<attribute name=[NAME]`
`type=[TYPE]`
`sdo:propertyType=[P_TYPE] />`
 	
where `[TYPE] = IDREF, IDREFS, anyURI` or restrictions of these types.
	`Property name=[NAME]`
	`type=[P_TYPE]`
	`many=true` (for IDREFS only)

Attribute with bi-directional property to a DataObject with sdo:oppositeProperty and sdo:propertyType
 	
`<attribute name=[NAME]`
`type=[TYPE]`
`sdo:propertyType=[P_TYPE]`
 	
`sdo:oppositeProperty=[PROPERTY] />`
 	
where:
`[TYPE] = IDREF, IDREFS, anyURI` or restrictions of these types.
	`Property name=[NAME]`
	`type=[P_TYPE]`
	`opposite=[PROPERTY]`
	`many=true` (for IDREFS only)
	
	Declared on:
	`Type [P_TYPE]:`
	`Property name=[PROPERTY]`
	`type=[NAME].containingType`
	`opposite=[NAME]`
	`containingType=[P_TYPE]`

Attribute with sdo:dataType
 	
`<attribute name=[NAME]`
`type=[TYPE]`
`sdo:dataType=[SDO_TYPE] />`
	`Property name=[NAME]`
	`type=[SDO_TYPE]`
	• The type of the property is the SDO type for [SDO_TYPE]
	• Used when the instance class for TYPE is not appropriate.

XML Global Elements and Attributes	SDO Property
Global Element `<element name=[NAME] />`	Same as local element declaration except the containing Type is not specified by SDO other than the Type's URI is the XSD target namespace.
Global Attribute `<attribute name=[NAME] />`	Same as local attribute declaration except the containing Type is not specified by SDO other than the Type's URI is the XSD target namespace.

Mapping of XSD Elements

If a ComplexType has content with two elements that have the same local name and the same targetNamespace, whether through declaration, extension, substitution, groups, or other means, the duplication is handled as follows:

- The ComplexType becomes a sequenced type, as if sdo:sequence="true" was declared.
- A single property is used for all the elements with the same local name and the same targetNamespace, where isMany=true.
- The type of the property is SDO Object.
- When substitution is possible for a Type, Type.open is true.

If schema extension is used, the base type may need to be modified with sdo:sequence="true" and elements with name conflicts introduced in extensions require that the property in the extended base type must be made isMany=true.

XML Elements	SDO Property
Element `<element name=[NAME] />`	`Property name=[NAME]`

Element with sdo:name
 	
`<element name=[NAME]`
`  sdo:name=[SDO_NAME] />`
	`Property name=[SDO_NAME]`

Element with sdo:aliasName
 	
`<element name=[NAME]`
`  sdo:aliasName=[ALIAS_NAME]`
`  type=[TYPE] />`
	`Property name=[NAME]`
	`  aliasName=[ALIAS_NAME]`
	`  type=[TYPE]`

Element reference
 	
`<element ref=[ELEMENT] />`
	`Property name=[ELEMENT].[NAME]`
	`type=[ELEMENT].[TYPE]`
	`default=[ELEMENT].[DEFAULT]`
	• Use the XSDHelper to determine the URI of the element if the referenced element is in another namespace.

Element with maxOccurs > 1
 	
`<element name=[NAME]`
`maxOccurs=[MAX] />`
 	
where `[MAX] > 1`
	`Property name=[NAME]`
	`many=true`

Element in all, choice, or sequence
 	
`<[GROUP] maxOccurs=[G_MAX]>`
`<element name=[NAME]`
`type=[TYPE]`
`maxOccurs=[E_MAX] />`
`</[GROUP] >`
 	
where
[GROUP] = all, choice, sequence
• Element groups and model groups are treated as if they were expanded in place.
• Nested [GROUP]s are expanded.
	`Property name=[NAME]`
	`type=[TYPE]`
	`many=true`
	
	Type sequenced=true
	
	• A property is created for every element
	• many=true when E_MAX or G_MAX is > 1
	• sequenced=true if the content allows elements to be interleaved. (for example <A/><B/><A/>)
	• sequenced=true if G_MAX > 1 and there is more than one element in this group or a contained group.
	• Property declarations are the same whether group is <all> or <choice> or <sequence>
	• Property behavior ignores group declarations.
	• Validation of DataObjects for the group constraints is external to the DataObject interface.

Element with nillable
 	
`<element name=[NAME]`
`nillable="true" type=[TYPE]/>`
	`Property name=[NAME]`
	
	• If the type of the element has Simple Content without attributes, a Java Type with an Object instance class is assigned. For example, IntObject instead of Int.
	• In an XML document, xsi:nil="true" corresponds to a null value for this property.

Element with substitution group
 	
`<element name=[NAME]`
`  type=[TYPE]`
`  substitutionGroup=[ELEMENT] />`
	`Property name=[NAME]`
	`  type=[TYPE]`
	
	• Use getInstanceProperties() for reflection.
	• Use get(property) to access the value.

Elements of Complex Type follow this table, in addition.

XML Elements with Complex Type	SDO Property

`<element name=[NAME]`
`  type=[TYPE] />`
	`Property name=[NAME]`
	`  type=[TYPE]`
	`  containment=true`

Elements of Simple Type follow this table, in addition.

XML Elements with Simple Type	SDO Property

Element of SimpleType
 	
`<element name=[NAME]`
`  type=[TYPE] />`
	`Property name=[NAME]`
	`  type=[TYPE]`
	
	• DataObject accessors may enforce simple type constraints.

Element of SimpleType with default
 	
`<element name=[NAME]`
`  type=[TYPE]`
`  default=[DEFAULT] />`
	`Property name=[NAME]`
	`  type=[TYPE]`
	`  default=[DEFAULT]`

Element of SimpleType with fixed
 	
`<element name=[NAME]`
`  type=[TYPE]`
`  fixed=[FIXED] />`
	`Property name=[NAME]`
	`  type=[TYPE]`
	`  default=[FIXED]`

Element of SimpleType with sdo:string
 	
`<element name=[NAME]`
`  type=[TYPE]`
`  sdo:string="true" />`
	`Property name=[NAME]`
	`  type=String`
	• The type of the property is SDO String
	• Used when the instance class for TYPE is not appropriate.

Element referencing a DataObject with sdo:propertyType
 	
`<element name=[NAME]`
`  type=[TYPE]`
`  sdo:propertyType=[P_TYPE] />`
 	
where `[TYPE] = IDREF, IDREFS, anyURI` or restrictions of these types
	`Property name=[NAME]`
	`  type=[P_TYPE]`
	`  many=true` (for IDREFS only)

Element with bi-directional reference to a DataObject with sdo:propertyType and sdo:oppositeProperty
 	
`<element name=[NAME]`
`  type=[TYPE]`
`  sdo:propertyType=[P_TYPE]`
 	
`sdo:oppositeProperty=[PROPERTY]`
`/>`
 	
where `[TYPE] = IDREF, IDREFS, anyURI` or restrictions of these types
	`Property name=[NAME]`
	`  opposite=[PROPERTY]`
	`  type=[P_TYPE]`
	`  many=true` (for IDREFS only)
	
	Declared on `Type PR_TYPE]:`
	`Property name=[PROPERTY]`
	`  type=[NAME].containingType`
	`  opposite=[NAME]`
	`  containingType=[P_TYPE]`

Element of SimpleType with sdo:dataType
 	
`<element name=[NAME]`
`  type=[TYPE]`
`  sdo:dataType=[SDO_TYPE] />`
	`Property name=[NAME]`
	`  type=[SDO_TYPE]`
	• The type of the property is the SDO type for [SDO_TYPE]
	• Used when the instance class for TYPE is not appropriate.

XML Schema Element special types	**SDO Property**

Element with type SDO ChangeSummaryType
 	
`<element name=[NAME]`
`  type="sdo:ChangeSummaryType"/>`
	`Property name=[NAME]`
	`  type=ChangeSummaryType`
	`  readOnly=true`

Mapping of XSD Built in Data Types

SDO specifies mappings from XSD to SDO Types.

A Java implementation may use an SDO Java Type if an Object wrapper for a primitive type is desirable. For example, XSD int maps to SDO Int, and an implementation may instead use SDO Java Type IntObject. The choice is made at the discretion of the implementation. The actual Type selected is set in property.type, enabling reflective access to the information.

The URI of the SDO Types is commonj.sdo. The instanceClass for each SDO Type is shown in the Java instance class column for convenience. If the XSD type of the instance value cannot be determined, or the SDO Type's instance class is java.lang.Object or null, the value is read as a String. AnySimpleType will read document values in as String unless xsi:type is specified in the document. Mixed text is read as SDO built in Property text from Type TextType.

List<String> on JDKs earlier than 1.5 are List with String entries.

XSD Simple Type	SDO Type	Java instance Class
anySimpleType	Object	java.lang.Object Values must support toString() for String value
anyType	DataObject	DataObject
anyURI	URI (override with sdo:propertyType)	String
base64Binary	Bytes	byte[]
boolean	Boolean	boolean or java.lang.Boolean
byte	Byte	byte or java.lang.Byte
date	YearMonthDay	String
dateTime	DateTime	String
decimal	Decimal	java.math.BigDecimal
double	Double	double or java.lang.Double
duration	Duration	String
ENTITIES	Strings	List<String>
ENTITY	String	String
float	Float	float or java.lang.Float
gDay	Day	String
gMonth	Month	String
gMonthDay	MonthDay	String
gYear	Year	String
gYearMonth	YearMonth	String
hexBinary	Bytes	byte[]

ID	String	String
IDREF	String (override with sdo: propertyType)	String
IDREFS	Strings (override with sdo: propertyType)	List<String>
int	Int	int or java.lang.Integer
integer	Integer	java.math.BigInteger
language	String	String
long	Long	long or java.lang.Long
Name	String	String
NCName	String	String
negativeInteger	Integer	java.math.BigInteger
NMTOKEN	String	String
NMTOKENS	Strings	List<String>
nonNegativeInteger	Integer	java.math.BigInteger
nonPositiveInteger	Integer	java.math.BigInteger
normalizedString	String	String
NOTATION	String	String
positiveInteger	Integer	java.math.BigInteger
QName	URI	String
short	Short	short or java.lang.Short
string	String	String
time	Time	String
token	String	String
unsignedByte	Short	short or java.lang.Short
unsignedInt	Long	long or java.lang.Long
unsignedLong	Integer	java.math.BigInteger
unsignedShort	Int	int or java.lang.Integer

The next table describes which XSD representation is used when writing Java instance objects as one of the following:

1. XML element.
2. Attribute values of type anySimpleType.
3. Union of SimpleTypes that have the same instance classes.

xsi:type is written for elements of type anySimpleType. Instance classes not in this table use XSD String as their type and toString() as their value.

Java instance Class	XSD Type
BigDecimal	decimal
BigInteger	integer
boolean or java.lang.Boolean	Boolean
byte or java.lang.Byte	byte
byte[]	hexBinary
char or java.lang.Character	string
Date	dateTime
double or java.lang.Double	double
float or java.lang.Float	float
int or java.lang.Integer	int
long or java.lang.Long	long
short or java.lang.Short	short
String or List<String>	string

Conversion between XSD QName and SDO URI

When an XML document is loaded, a value of type xsd:QName is converted into an SDO URI with a value of:

- The namespace name + # + local part

where + indicates string concatenation.

When an XML document is saved, a value of type SDO can be converted back to an xsd:Qname, if that is the expected XML type:

- The URI value is parsed into two parts:
  - The namespace name is the URI up to but not including the last # character in the URI value.
  - The local part is the URI after the last # character in the URI value.
- An XML namespace declaration for a namespace prefix is made in the XML document. The declaration may be made at any enclosing point in the document in an implementation-dependent manner or an existing declaration may be reused.
- The declaration is of the form xmlns:prefix="namespace name".
- The prefix is implementation-dependent.
- The QName value is of the form prefix:local part.

Example:

- Message is a property of XSD type QName and SDO type URI
- Load: <input message="tns:inputRequest" name="inputMessage" xmlns:tns="http://example.com" />
- inputDataObject.get(message) returns http://example.com#inputRequest
- inputDataObject.set(message, "http://test.org#testMessage")
- Save: <input message="tns:testMessage" name="inputMessage" xmlns:tns="http://test.org" />

Dates

Considering the importance of Date information, it is unfortunate that there are few good software standards for handling this information.

SDO chose java.util.Date and java.lang.String as the instance classes for Date types because they are the simplest classes sufficient to enable technology-independent scenarios. java.util.Date is effectively a simple numeric value without behavior, a concept that is widely used as the underlying indicator of absolute time across languages and operating systems. The string representations are from XML Schema and easy to convert to other representations.

Operating on Date values, such as applying calendar, time zone, order, duration, and locale settings, is best left to helper and utility classes, such as GregorianCalendar, XMLGregorianCalendar, and SimpleDateFormat. The implementation cost of java.util.Date and java.lang.String is far lower than the calendar classes, which have more fields than most of the DataObjects that will contain them. In the case where Date and java.lang.String are insufficient, sdo:dataType can be used to override the datatype to one with a custom implementation class.

Examples of XSD to SDO Mapping

XSD	SDO

Schema declaration
 	
`<schema targetNamespace= "http://`
`www.example.com/IPO">`
	`uri="http://www.example.com/IPO"`

Global Element with Complex Type
 	
`<element name="purchaseOrder"`
`type="PurchaseOrderType"/>`
	`Property name="purchaseOrder"`
	`type="PurchaseOrderType"`
	`containment=true`

Global Element with Simple Type
 	
`<element name="comment"`
`type="xsd:string"/>`
	`Property name="comment" type="sdo:String"`

Complex Type
 	
`<complexType`
`name="PurchaseOrderType">`
	`Type name="PurchaseOrderType" uri="http:/`
	`/www.example.com/IPO"`

Simple Type
 	
`<simpleType sdo:name="QuantityType">`
`<restriction`
`base="positiveInteger">`
`<maxExclusive value="100"/>`
`</restriction>`
`</simpleType>`
 	
`<simpleType name="SKU">`
`<restriction base="string">`
`<pattern value="\d{3}-[A-Z]{2}"/>`
`</restriction>`
`</simpleType>`
	`Type name="QuantityType"`
	`dataType=true`
	`base="sdo:Int" uri="http://`
	`www.example.com/IPO"`
	
	`Type name="SKU" instanceClass="String"`
	`dataType=true uri="http://`
	`www.example.com/IPO"`
	`base="sdo:String"`

Local Element with Complex Type
 	
`<element name="shipTo"`
`type="ipo:Address"/>`
 	
`<element name="billTo"`
`type="ipo:Address"/>`
 	
`<element name="items"`
`type="ipo:Items"/>`
	`Property name="shipTo" type="Address"`
	`containment=true`
	`containingType="PurchaseOrderType"`
	
	`Property name="billTo" type="Address"`
	`containment=true`
	`containingType="PurchaseOrderType"`
	
	`Property name="items" type="Items"`
	`containment=true`
	`containingType="PurchaseOrderType"`

Local Element with Simple Type
 	
`<element ref="ipo:comment"`
`minOccurs="0"/>`
 	
`<element name="productName"`
`type="string"/>`
	`Property name="comment" type="String"`
	`containingType="PurchaseOrderType"`
	
	`Property name="productName" type="String"`
	`containingType="Items"`

Local Attribute
 	
`<attribute name="orderDate"`
`type="date"/>`
 	
`<attribute name="partNum"`
`type="ipo:SKU" use="required"/`
`>`
	`Property name="orderDate" type="Date"`
	`containingType="PurchaseOrderType"`
	
	`Property name="partNum" type="SKU"`
	`containingType="ItemType"`

Type extension
 	
`<complexType name="USAddress">`
`<complexContent>`
`<extension base="ipo:Address">`
	`Type name="USAddress" uri="http://`
	`www.example.com/IPO"`
	`base="ipo:Address"`

Local Attribute fixed value declaration
 	
`<attribute name="country"`
`type="NMTOKEN" fixed="US"/>`
	`Property name="country" type="String"`
	`default="US"`
	`containingType="USAddress"`

Multi-valued local element declaration
 	
`<element name="item" minOccurs="0"`
`maxOccurs="unbounded">`
`<complexType sdo:name="ItemType"/>`
`</element>`
	`Property name="item" type="ItemType"`
	`containment=true many=true`
	`containingType="Items"`
	
	`Type name="ItemType" uri="http://`
	`www.example.com/IPO"`

Attribute reference declarations
 	
`<attribute name="customer"`
`type="IDREF"`
`sdo:propertyType="cust:Customer"`
`sdo:oppositeProperty="purchaseOrder" />`
 	
`<attribute name="customer"`
`type="anyURI"`
`sdo:propertyType="cust:Customer"/>`
 	
`<attribute ref="xlink:href"`
`sdo:propertyType="cust:Customer" sdo:name="customer" />`
	`Property name="customer" type="Customer"`
	`opposite="Type[name='Customer']/`
	`property[name='purchaseOrder']"`
	`containingType="PurchaseOrderType"`
	
	Declared in the Customer type:
	`Property name="purchaseOrder"`
	`type="PurchaseOrderType"`
	
	`opposite="Type[name='PurchaseOrderType']/ property[name='customer']"`
	`containingType="Customer"`
	
	`Property name="customer" type="Customer"`
	`containingType="PurchaseOrderType"`
	
	`Property name="customer" type="Customer"`
	`containingType="PurchaseOrderType"`

Local Attribute ID declaration
 	
`<attribute name="primaryKey"`
`type="ID"/>`
	`Property name="primaryKey" type="String"`
	`containingType="Customer"`

Local Attribute default value declaration
 	
`<xsd:attribute name="country"`
`type="xsd:NMTOKEN"`
`default="US"/>`
	`Property name="country" type="String"`
	`default="US"`
	`containingType="USAddress"`

Abstract ComplexTypes
 	
`<complexType name="Vehicle"`
`abstract="true"/>`
	`Type name="Vehicle"`
	`abstract=true uri="http://`
	`www.example.com/IPO"`

SimpeType unions
 	
`<xsd:simpleType name="zipUnion">`
`<xsd:union`
`memberTypes="USState`
`listOfMyIntType"/>`
`</xsd:simpleType>`
	Type SDO Object is used as the Type for every Property resulting from elements and attributes with SimpleType zipUnion.

Notes:

1. Examples are from, or based on, IPO.xsd in http://www.w3.org/TR/xmlschema-0/
2. Type[name='Customer']/property[name='purchaseOrder'] refers to the declaration of the purchaseOrder Property in the Type Customer in the same document.

Example of SDO annotations

This example shows the use of sdo:string, sdo:dataType, sdoJava:package, and sdoJava:instanceClass

```
<schema targetNamespace="http://www.example.com/IPO"
        xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:ipo="http://www.example.com/IPO"

        xmlns:sdo="commonj.sdo/XML"
        xmlns:sdoJava="commonj.sdo/java"
        sdoJava:package="com.example.myPackage">


 <complexType name="PurchaseOrderType" >
  <sequence>
   <element name="shipTo"     type="ipo:Address"/>
   <element name="billTo"     type="ipo:Address"/>
   <element ref="ipo:comment" minOccurs="0"/>
   <element name="items"      type="ipo:Items"/>
  </sequence>
  <attribute name="orderDate" type="date" sdo:dataType="ipo:MyGregorianDate"/>
 </complexType>

 <complexType name="Items">
  <sequence>
   <element name="item" minOccurs="0" maxOccurs="unbounded">
    <complexType sdo:name="Item">
     <sequence>
      <element name="productName" type="string"/>
      <element name="quantity" sdo:dataType="sdo:Int">
       <simpleType>
        <restriction base="positiveInteger">
         <maxExclusive value="100"/>
        </restriction>
       </simpleType>
      </element>
      <element name="USPrice"    type="decimal"/>
      <element ref="ipo:comment" minOccurs="0" sdo:aliasName="itemComment"/>
```

```
      <element name="shipDate"   type="date" minOccurs="0" sdo:string="true"/>
     </sequence>
     <attribute name="partNum" type="ipo:SKU" use="required"/>
    </complexType>
   </element>
  </sequence>
 </complexType>

 <simpleType name="MyGregorianDate"
sdoJava:instanceClass="java.xml.datatype.GregorianCalendar"/>

 <simpleType name="SKU" sdoJava:instanceClass="com.example.SKU">
  <restriction base="string">
   <pattern value="\d{3}-[A-Z]{2}"/>
  </restriction>
 </simpleType>

</schema>
```

XML use of Sequenced Data Objects

Sequenced Data Objects are DataObjects with a sequence capturing the additional XML order information that is specific to XML documents.

Sequenced DataObjects have Type.sequenced=true. The XSD to SDO mapping defines an XML DataObject to be used when sdo:sequence="true" is declared in the XSD type.

The XML use of Sequenced DataObject defines a Sequence returned from the DataObject interface:

- getSequence() - A Sequence of all the elements and mixed text in the content of an XML element. Each entry in the Sequence represents either one XML element designated by the entry's Property, or XML mixed text, designated by the SDO Text Property. The name of the property is the same as the name of the XML element unless sdo:name was used to replace the name. The values of the entries are available through both the Sequence API and the DataObject API for the Properties. DataObject.getInstanceProperties() includes all the Properties in the Sequence. For open content, where XML any declarations, substitution groups or mixed content were used, the Properties of the entries might be declared in other Types than the DataObject's Type. The order of the entries in the Sequence is the same as the order of XML elements.

XSD Mapping Details

The following guidelines apply when mapping XSD to SDO:

1. The order of the Properties are declared within a Type is the lexical order of their declaration in an XSD. All Properties of the Type extended precede local declarations within the Type.
2. The XSD names are preserved in the Type and Property. Use the sdo:name override to modify names as an option to remove duplicate names, blank names, or names with special characters.
3. All declarations not covered in this Mapping may be ignored by a compliant implementation.
4. All <group> references, <attributeGroup> references, <include>s, and <import>s are fully expanded to the equivalent XSD as if these declarations were not present.
5. <choice> declarations for Complex Content are treated as <sequence> for the purpose of declaring Properties.
6. All comments, processing instructions, and annotations other than appinfo are discarded to the equivalent XSD as if these declarations were not present.
7. Redefinitions are expanded to the equivalent XSD as if these declarations were not present.
8. Model Groups (sequence, all, choice, group) do not contribute to the mapping except for maxOccurs>1 results in Properties with many=true.
9. Global group and attribute group declarations that include type declarations follow the normal mapping rules for those type declarations. The same types are used in all places the groups are referenced.

Compliance

The mappings here are the base mappings. Vendors may extend the mappings provided that client programs developed against these mappings continue to run. An SDO program using this mapping, and the DataObject, should be portable across vendor-added mappings and implementations.

Importing the sdo:alias annotation for XSDs is optional.

Corner cases

This specification does not standardize the mapping for corner cases. We follow the principle that complexity is never added to the simple cases to handle these more advanced cases. Future versions of SDO may define mappings for these corner cases.

1. List of lists without unions.
2. <element nillable="true" maxOccurs="unbounded" type="USAddress"/> Multi-valued nillable Properties with DataObject Types.
3. key and keyref.

4. When an element of anyType is used with xsi:type specifying simple content, a wrapper DataObject must be used with a property named "value" and type of SDO Object that is set to the wrapped type. For example, <element name="e" type="anyType"> and a document <e xsi:type="xsd:int">5</e> results in a wrapper DataObject where the value property is set to the Integer with value 5.
5. In some cases it is not possible to maintain an SDO base relationship when one exists in schema. This can happen for example when complex types extend simple types or when sdoJava:instanceClass is specified.
6. Elements that occur more than once and have type IDREFS and have sdo:propertyType will not be able to distinguish between consecutive elements in an XML document and one element with all the values in a single element. If there are interleaving elements sequence must be true to distinguish the order between elements. XML Schema recommends against the use of elements with type IDREF or IDREFS.
7. Anonymous type declarations in global group declarations, which are not a recommended schema design practice.

XML without Schema to SDO Type and Property

When a document does not have a schema, the following table and principles define the Types and Properties.

1. Each URI defines a DocumentRoot Type that contains all Properties that occur in the same URI.
2. Instances of these DocumentRoot Types will accept any Property in DataObject.get(property) and set(property).
3. If two XML elements or attributes have the same URI and Name, they are mapped to the same Property.
4. The URI is determined by the xmlns declarations in the document.
5. Mixed text is mapped to the SDO text Property.
6. The values allowed for Properties for XML attributes may not be DataObject.

XML Document	SDO Type and Property

XML elements
 	
`<[URI]:[NAME]>`
`...`
`</[URI]:[NAME]>`
	`Type name="DocumentRoot" sequenced=true`
	`uri=[URI] open=true`
	
	contains:
	
	`Property name=[NAME] type=DataObject`
	`containment=true many=true`
	`containingType=DocumentRoot`

XML attributes
 	
`<element`
`[URI]:[NAME]="value"/>`
	`Type name="DocumentRoot" sequenced=true`
	`uri=[URI] open=true`
	
	contains:
	
	`Property name=[NAME] type=string`
	`containingType=DocumentRoot`

Generation of XSD from SDO Type and Property

When SDO Types and Properties did not originate from an XSD definition, it is often useful to define the equivalent XML schema declarations.

When an XSD is generated from Type and Property it contains all the information defined in the SDO Model. An XSD generated from Type and Property will round trip back to the original Type and Property. However, if the XSD was not generated and is used to create the Type and Property, regenerating the XSD will not round trip to produce the original. This is because there is more information in an XSD than in Type and Property, primarily focused on defining the XML document syntax.

The mapping principles are summarized in this table. A URI defines a schema and a target namespace. An SDO Type defines an XSD complex type and a global element declaration. An SDO property defines either a local element or an attribute in a complex type.

SDO	XSD
URI	<schema targetNamespace>
Type	<complexType> <element> global // or <simpleType>
Property	<element> local // or <attribute>

Each XSD contains Types with the same URI. When referring to other ComplexTypes, the implementation is responsible for generating the appropriate import and include XSD declarations.

An XSD can only be generated when:

1. Multiple inheritance is not used.
   - That is, all Types have no more than 1 base in Types.getBaseTypes().
2. The names of the Types and Properties are valid XSD identifiers.

The following defines the minimal XML schema declarations. When opening XML elements are shown the appropriate ending XML element is produced by the implementation. An implementation may include additional declarations as long as documents that validate with the

generated schema also generate with the customized schema. In addition, an implementation is expected to generate all required namespace declarations, includes, and imports necessary to produce a valid XML schema.

If a namespace declaration shown in the generation templates is not used by the XSD, it may be suppressed. Namespace declarations may have prefix names chosen by the implementation (instead of xsd, sdo, sdoJava, and tns). The specific elements containing the namespace declarations are determined by the implementation.

The schemas generated are a subset of the XMI 2.0 and 2.1 specifications. It is permissible to generate the xmi:version attribute from the XMI specification to enable XMI conformant software to read the XSDs and valid XML documents.

The Schema element itself is generated with a target namespace determined by the URI of the Types that will be defined in the schema. If the Types have a javaPackage specified then the sdo:JavaPackage attribute is present in the schema declaration.

- `[URI]` is defined by type.uri. If [URI] is null then the XSD is generated without a targetNamespace.
- `[JAVA_PACKAGE]` is defined by Type.getInstanceClass().getPackage().toString().

SDO	**XSD Schema**

	`<xsd:schema`
	`xmlns:xsd="http://www.w3.org/2001/XMLSchema"`
	`xmlns:sdo="commonj.sdo"`
	`xmlns:sdoJava="commonj.sdo/java">`

`[URI]`	`xmlns:tns=[URI]` `targetNamespace=[URI]`
`[JAVA_PACKAGE]`	`sdoJava:package=[JAVA_PACKAGE]`

For each Type that is a dataType, type.dataType==true, an XSD SimpleType is generated. The SimpleType is based on the following:

- [NAME] is type.name
- [ABSTRACT] is type.abstract.
- [ALIAS_NAME] is space separated values from type.aliasNames and is produced if there are alias names.
- [JAVA_INSTANCE_CLASS] is type.getInstanceClass().getName() and is produced if not null.
- [BASE.NAME] is the name of the base type, type.getBaseTypes().get(0).getName() if not null. When not null, the simple type extends the base type. tns: is the prefix for the URI of the base type, type.getBaseTypes().get(0).getURI(). If the base type is in another namespace the appropriate namespace and import declarations are produced by the

implementation. If there are no base types, then the xsd type used is from the table "Mapping of SDO DataTypes to XSD Built in Data Types" based on the instance class.

SDO Type	**XSD SimpleType**
	`<xsd:simpleType name=[NAME]>`
`[ABSTRACT]`	`  abstract="true"`
`[ALIAS_NAME]`	`  Sdo:aliasName=[ALIAS_NAME]`
`[JAVA_INSTANCE_CLASS]`	`  sdoJava:instanceClass=[JAVA_INSTANCE_CLASS]`
`[BASE.NAME]`	`  <xsd:restriction base=tns:[BASE.NAME]>`

For each Type that is not a dataType, type.dataType==false, an XSD ComplexType and a global element is generated. The ComplexType is based on the following:

- [NAME] is type.name
- [ABSTRACT] is type.abstract.
- [ALIAS_NAME] is space separated values from type.aliasNames and is produced if there are alias names.
- [BASE.NAME] is the name of the base type, type.getBaseTypes().get(0).getName() and is produced if not null. When produced, the complex type extends the base type. tns: is the prefix for the URI of the base type, type.getBaseTypes().get(0).getURI(). If the base type is in another namespace the appropriate namespace and import declarations are produced by the implementation.
- [SEQUENCED] indicates if the type is sequenced, type.sequenced. If true, the complex type declaration is mixed and the content of the element is placed in a <choice>. If false, the complex type contents are placed in a <sequence>. If no local elements are generated, the <choice> or <sequence> is suppressed.
- [OPEN] indicates if the type accepts open content, type.open. An <any> is placed in the content and <anyAttribute> is placed after the content.

SDO Type	XSD ComplexType
	`<xsd:complexType name=[NAME]>`
`[ABSTRACT]`	`  abstract="true"`
`[ALIAS_NAME]`	`  sdo:aliasName=[ALIAS_NAME]`
`[BASE.NAME]`	`  <xsd:complexContent>` `    <xsd:extension base=tns:[BASE.NAME]>`
`[SEQUENCED]`	`      mixed="true"` `      <xsd:choice maxOccurs="unbounded">`
`![SEQUENCED]`	`      <xsd:sequence>`
`[OPEN]`	`        <xsd:any maxOccurs="unbounded" processContents="lax"/>` `      <xsd:anyAttribute processContents="lax"/>`

The global element for the type:

- lowercase(TYPE.NAME) is the type name with the first letter converted to lower case as defined type java.lang.Character.toLowerCase(). If two global elements with the same name and target namespace would be generated when the lowercase is applied, then the original type name is used unchanged.
- [TYPE.NAME] is the type name type.name.

SDO Type	XSD Global Element
	`<xsd:element name=[lowercase(TYPE.NAME)] type=tns:[TYPE.NAME] />`

For each property in type.getDeclaredProperties(), either an element or an attribute will be generated, declared within the content of the property's containing type property.getContainingType(). An element is generated if either property.many or property.containment is true, or if property.get(xmlElement) is present and set to true. xmlElement is a property from XMLInfo. If the property is bidirectional and the opposite property has containment=true, nothing is generated. Otherwise, an attribute is generated. Round-trip between SDO models and their generated XSDs will preserve the order of the properties when all elements are generated.

- [NAME] is property.name
- [ALIAS_NAME] is space separated values from property.aliasNames and is produced if there are alias names.
- [READ_ONLY] is the value of property.readOnly and is produced if true.
- [MANY] indicates if property.many is true and maxOccurs is unbounded if true.
- [CONTAINMENT] indicates if property.containment is true.
    - When containment is true, then DataObjects of that Type will appear as nested elements in an XML document.
    - When containment is false and the property's type is a DataObject, a URI reference to the element containing the DataObject is used and an sdo:propertyType declaration records the target type. Values in XML documents will be of the form "#xpath" where the xpath is an SDO DataObject XPath subset. It is typical to customize the declaration to IDREF if the target element has an attribute with type customized to ID.
    - [TYPE.NAME] is the type of the element. If property.type.dataType is true, [TYPE.NAME] is the name of the XSD built in SimpleType corresponding to property.type, where the prefix is for the xsd namespace. Otherwise, [TYPE.NAME] is property.type.name where the tns: prefix is determined by the namespace declaration for the Type's URI.
- [OPPOSITE.NAME] is the opposite property if the property is bidirectional and indicated when property.opposite is not null.

SDO Property	XSD Element
	<xsd:element name=[NAME] minOccurs="0"
[ALIAS_NAME]	Sdo:aliasName=[ALIAS_NAME]
[READ_ONLY]	sdo:readOnly=[READ_ONLY]
[MANY]	maxOccurs="unbounded"
[CONTAINMENT]	type="tns:[TYPE.NAME]"
![CONTAINMENT]	type="xsd:anyURI" sdo:propertyType="tns:[TYPE.NAME]"
[OPPOSITE.NAME]	sdo:oppositeProperty=[OPPOSITE.NAME]

For all the properties in type.getDeclaredProperties() where the element test rules above indicate that an attribute is generated, a local attribute declaration is produced.

- [NAME] is property.name
- [ALIAS_NAME] is space separated values from property.aliasNames and is produced if there are alias names.
- [READ_ONLY] is the value of property.readOnly and is produced if true.
- [DEFAULT] is property.default and is produced if the default is not null and the default differs from the XSD default for that data type .
- [TYPE.DATATYPE] indicates if property.type.dataType is true.
    - When isDataType is true, [TYPE.NAME] is the name of the XSD built in SimpleType corresponding to property.type, where the prefix is for the xsd namespace.
    - When isDataType is false, [TYPE.NAME] is property.type.name where the tns: prefix is determined by the namespace declaration for the Type's URI. A URI reference to the element containing the DataObject is used and an sdo:propertyType declaration records the target type. Values in XML documents will be of the form "#xpath" where the xpath is an SDO DataObject XPath. It is typical to customize the declaration to IDREF if the target element has an attribute with type customized to ID.
- [OPPOSITE.NAME] is the opposite property if the property is bidirectional and indicated when property.opposite is not null.

SDO Property	XSD Attribute
	<xsd:attribute name=[NAME]
[ALIAS_NAME]	sdo:aliasName=[ALIAS_NAME]
[READ_ONLY]	sdo:readOnly=[READ_ONLY]
[DEFAULT]	default=[DEFAULT]
[TYPE.DATATYPE]	type="tns:[TYPE.NAME]"
![TYPE.DATATYPE]	type="xsd:anyURI" sdo:propertyType=tns:[TYPE.NAME]
[OPPOSITE.NAME]	sdo:oppositeProperty=[OPPOSITE.NAME]

Mapping of SDO DataTypes to XSD Built in Data Types

For the SDO Java Types, the corresponding base SDO Type is used. For the SDO Java Types, and for SDO Date, an sdo:dataType annotation is generated on the XML attribute or element referring to the SDO Type.

SDO Type	XSD Type
Boolean	boolean
Byte	byte
Bytes	hexBinary
Character	string
DataObject	anyType
Date	dateTime
DateTime	dateTime
Day	gDay
Decimal	decimal
Double	double
Duration	duration
Float	float
Int	int
Integer	integer
Long	long
Month	gMonth
MonthDay	gMonthDay
Object	anySimpleType
Short	short
String	string
Strings	string
Time	time
Year	gYear
YearMonth	gYearMonth
YearMonthDay	date
URI	anyURI

Example Generated XSD

If the Types and Properties for the PurchaseOrder schema had not come originally from XSD, then these rules would produce the following XML Schema.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">

<xsd:element name="purchaseOrder" type="PurchaseOrder"/>
 <xsd:complexType name="PurchaseOrder">
  <xsd:sequence>
   <xsd:element name="shipTo" type="USAddress" minOccurs="0"/>
```

```
   <xsd:element name="billTo" type="USAddress" minOccurs="0"/>
   <xsd:element name="items"  type="Items" minOccurs="0"/>
  </xsd:sequence>
  <xsd:attribute name="comment" type="xsd:string"/>
  <xsd:attribute name="orderDate" type="xsd:date"/>
 </xsd:complexType>

 <xsd:element name="uSAddress" type="USAddress"/>
 <xsd:complexType name="USAddress">
  <xsd:attribute name="name"   type="xsd:string"/>
  <xsd:attribute name="street" type="xsd:string"/>
  <xsd:attribute name="city"   type="xsd:string"/>
  <xsd:attribute name="state"  type="xsd:string"/>
  <xsd:attribute name="zip"    type="xsd:decimal"/>
  <xsd:attribute name="country" type="xsd:string" default="US"/>
 </xsd:complexType>

 <xsd:element name="items" type="Items"/>
 <xsd:complexType name="Items">
  <xsd:sequence>
   <xsd:element name="item" minOccurs="0" maxOccurs="unbounded"/>
  </xsd:sequence>
 </xsd:complexType>

 <xsd:element name="item" type="Item"/>
 <xsd:complexType name="Item">
  <xsd:attribute name="productName" type="xsd:string"/>
  <xsd:attribute name="quantity" type="quantityType">
  <xsd:attribute name="partNum"  type="SKU"/>
  <xsd:attribute name="USPrice"  type="xsd:decimal"/>
  <xsd:attribute name="comment"  type=xsd:string"/>
  <xsd:attribute name="shipDate" type="xsd:date"/>
 </xsd:complexType>

 <xsd:simpleType name="quantityType">
  <xsd:restriction base="xsd:int"/>
 </xsd:simpleType>

 <xsd:simpleType name="SKU">
  <xsd:restriction base="xsd:string"/>
 </xsd:simpleType>

</xsd:schema>
```

The following is the serialization of the example purchase order that matches this schema.

```
<?xml version="1.0"?>
<purchaseOrder xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```
    orderDate="1999-10-20" comment="Hurry, my lawn is going wild!">
  <shipTo country="US" name="Alice Smith" street="123 Maple Street"
    city="Mill Valley" state="CA" zip="90952"/>
  <billTo country="US" name="Robert Smith" street="8 Oak Avenue"
    city="Old Town" state="PA" zip="95819"/>
  <items>
    <item partNum="872-AA" productName="Lawnmower"
      quantity="1" USPrice="148.95" comment="Confirm this is electric"/>
    <item partNum="926-AA" productName="Baby Monitor"
      quantity="1" USPrice="39.98" shipDate="1999-05-21"/>
  </items>
</purchaseOrder>
```

Customizing Generated XSDs

Because an XSD contains more information than Type and Property, there are many XSD capabilites unused by the default generation, for example the preference between serializing with XML elements or attributes. The recommended procedure is to generate the XSD from Types and Properties, customize the XSD using tools or with XSLT, and use the customized XSD as the original from which to define the SDO Types and Properties.

DataGraph XML Serialization

A DataGraph may be serialized as an XML stream. If the Types and Properties came from XML Schema, the DataObjects are serialized following the XSD. If the metadata comes from another source, a virtual SDO XSD is generated and the DataObjects are serialized following the XSD.

The DataGraph's rootObject is a DataObject with exactly one property set. The name of this property is the root element name. The value of this property is the DataObject serialized in the root element. The DataGraph and the rootObject are two extra objects when compared to using DataObjects and XMLHelper. For example, for the purchase order XSD, a DataGraph's rootObject is a document DataObject with a containment property called "purchaseOrder" that contains the actual purchase order DataObject. Applications that do not use DataGraphs just use the purchase order DataObject directly and do not create the DataGraph and the document DataObject.

In general, the DataGraph serialization consists of a description of the schema used for the DataGraph, followed by the DataObjects that are contained in the DataGraph, followed by a description of the changes. The serialization of DataObjects follows the XMI specification or the XSD for the DataObject model, producing the same XML stream independent of the enclosing DataGraph element. When XML Schema is used as the metadata, the XML serialization of the DataObjects follows the XSD and the resulting XML elements should validate with the XML Schema when all the constraints for the XSD are enforced.

The description of the schema is optional and can be expressed either as an XSD or EMOF model. The description of the changes is also optional. The changes are expressed as a change summary. XSDs and models are typically included if it is likely that the reader of the DataGraph would not be able to retrieve the model by the logical URI of the XSD targetNamespace or EMOF Package URI. The serialization of the EMOF models follows the XMI specification. The optional serialization of the ChangeSummary also follows XMI, where properties that have not changed value are omitted. When serializing XSDs and models, only the XSDs and models actually used by the DataObjects are typically transferred. When the DataGraph was originally created from an XSD, the XSD form is preferred in order to preserve all original XSD information. If the DataGraph is from a source other than XSD, an XSD may be generated (typically following the EMOF and XMI specifications) and included, or the EMOF model may be included. The choice of which to include is determined by the serializer of the DataGraph.

The serialization of a DataGraph, whether invoked through a DAS or java.io.Serializable or in a Web service, is expected to be the same XML format described here. When a DataGraph is serialized in Java serialization, it is preceded by an int indicating the number of bytes in the DataGraph XML. When a single DataObject from a DataGraph is serialized, the format is an XPath subset of the DataObject’s path location within the DataGraph from the root, preceded by an int for the number of bytes in the XPath, and followed by the serialization of the DataGraph.

The XSD for the DataGraph serialization is:

```
<xsd:schema
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:sdo="commonj.sdo"
  targetNamespace="commonj.sdo">

  <xsd:element name="datagraph" type="sdo:DataGraphType"/>

  <xsd:complexType name="DataGraphType">
    <xsd:complexContent>
      <xsd:extension base="sdo:BaseDataGraphType">
        <xsd:sequence>
          <xsd:any minOccurs="0" maxOccurs="1" namespace="##other"
processContents="lax"/>
        </xsd:sequence>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>

  <xsd:complexType name="BaseDataGraphType" abstract="true">
    <xsd:sequence>
      <xsd:element name="models" type="sdo:ModelsType" minOccurs="0"/>
      <xsd:element name="xsd" type="sdo:XSDType" minOccurs="0"/>
      <xsd:element name="changeSummary" type="sdo:ChangeSummaryType"
minOccurs="0"/>
    </xsd:sequence>
    <xsd:anyAttribute namespace="##other" processContents="lax"/>
  </xsd:complexType>

  <xsd:complexType name="ModelsType">
    <xsd:annotation>
      <xsd:documentation>
        Expected type is emof:Package.
      </xsd:documentation>
    </xsd:annotation>
    <xsd:sequence>
      <xsd:any minOccurs="0" maxOccurs="unbounded" namespace="##other"
processContents="lax"/>
    </xsd:sequence>
  </xsd:complexType>

  <xsd:complexType name="XSDType">
    <xsd:annotation>
      <xsd:documentation>
        Expected type is xsd:schema.
      </xsd:documentation>
```

```
    </xsd:annotation>
    <xsd:sequence>
      <xsd:any minOccurs="0" maxOccurs="unbounded" namespace="http://
www.w3.org/2001/XMLSchema" processContents="lax"/>
    </xsd:sequence>
  </xsd:complexType>

  <xsd:complexType name="ChangeSummaryType">
    <xsd:sequence>
      <xsd:any minOccurs="0" maxOccurs="unbounded" namespace="##any"
processContents="lax"/>
    </xsd:sequence>
    <xsd:attribute name="create" type="xsd:string"/>
    <xsd:attribute name="delete" type="xsd:string"/>
    <xsd:attribute name="logging" type="xsd:boolean"/>
  </xsd:complexType>

  <xsd:attribute name="ref" type="xsd:string"/>

</xsd:schema>
```

Examples of this serialization can be seen in Accessing DataObjects using XPath subset and in Appendix – Complete DataGraph Serialization.

XPath Expression for DataObjects

Many of the accessor methods for DataObjects make use of a String parameter that provides the path that identifies the property to which the method applies.

The XPath expression is an augmented subset of XPath 1.0 [5] with the additional ability to access data using 0 as a base index, a style common throughout Java programming. Arrays and List.get(index) in Java both index from 0, and the intent is to enable the most productive environment for the Java programmer, avoiding the need for adding or subtracting 1 when using path expressions and Java indices together. The syntax for specifying these paths, is shown here:

```
path ::= (scheme ':')? '/'? (step '/')* step
scheme ::= [^:]+ ':'
step ::= '@'? property
      | property '[' index_from_1 ']'
      | property '.' index_from_0
      | reference '[' attribute '=' value ']'
      | ".."
property ::= NCName      ;; may be simple or complex type
attribute ::= NCName     ;; must be simple type
reference :: NCName      ;; must be DataObject type
index_from_0 ::= Digits
index_from_1 ::= NotZero (Digits)?
value ::= Literal
      | Number
      | Boolean
Literal ::= '"' [^"]* '"'
      | "'" [^']* "'"
Number ::= Digits ('.' Digits?)?
      | '.' Digits
Boolean ::= true
      | false
NotZero ::= [1-9]
Digits ::= [0-9]+

;; leading '/' begins at the root
;; ".." is the containing DataObject, using containment properties
;; Only the last step have an attribute as the property
```

The presence or absence of the @ sign in a path has no meaning. Properties are always matched by name independent of their XML representation.

The scheme is an extension mechanism for supporting additional path expressions in the future. No schema and a scheme of "sdo:" are equivalent, representing this syntax.

For example, consider the Company model described in “Complete Data Graph for Company Example” on page 140. One way to construct an XPath that can be used to access a DataObject contained in another DataObject is to specify the index of the contained DataObject within the

appropriate property. For example, given an instance of a Company DataObject called "company" one way to access the Department at index 0 in the "departments" list is:

```
DataObject department = company.getDataObject("departments.0");
```

Another way to access a contained DataObject is to identify that object by specifying the value of one of the attributes of that object. So, for example, given a Department DataObject called "department", one way to access the Employee where the value of the "SN" attribute is "E0002" is:

```
DataObject employee =
    department.getDataObject("employees[SN='E0002']");
```

It is also possible to write a path expression that traverses one or more references in order to find the target object. The two accesses shown above can be combined into a single call that gets the Employee using a path expression that starts from the company DataObject, for example

```
DataObject employee =
    company.getDataObject("departments.0/employees[SN='E0002']");
```

If more than one property shares the same name, only the first is matched by the path expression, using property.name for name matching. If there are alias names assigned, those are also used to match. Also, names including any of the special characters of the syntax (./[]="'@) are not accessible. Each step of the path before the last must return a single DataObject. When the property is a Sequence, the values returned are those of the getValue() accessor.

ChangeSummary XML format

The serialization of the ChangeSummary includes enough information to reconstruct the original information of the DataObjects at the point when logging was turned on. The goal of this format is to provide a simple XML representation that can express the difference between the graph when logging began and ended. The serialization of the state when logging is ended is the complete XML as serialized from XMLHelper and is referred to as the final XML in this section to contrast with the changeSummary XML.

DataObjects which are currently in the data graph, but were not present when logging was started are indicated in the change summary with a create attribute:

```
<changeSummary create="E0004" >
</changeSummary>
...
<employees name="Al Smith" SN="E0004"/>
...
```

Similarly, DataObjects deleted during logging are flagged with the "delete" attribute. In this case the change summary also contains a deep copy of the object which was deleted, as it no longer appears in the data graph. Also, the position in the tree must be recorded, so the departments property is reproduced, where there is an employees element for each employee object. The sdo:ref attribute is used to indicate the corresponding object that is represented in both the changeSummary and the final document. For example, `<employees sdo:ref="E0001"/>` refers to the employee with ID E0001 in the final document, `<employees name="John Jones" SN="E0001"/>`. The example below shows that the deleted employee has ID E0002, is located in the first department at the second position. The first and third employees are unchanged and the fourth employee is added.

```
<sdo:datagraph xmlns:company="company.xsd"
               xmlns:sdo="commonj.sdo">

  <changeSummary create="E0004" delete="E0002">
    <company sdo:ref="#/company" name="ACME" employeeOfTheMonth="E0002"/>
    <departments sdo:ref="#/company/departments[1]">
      <employees sdo:ref="E0001"/>
      <employees name="Mary Smith" SN="E0002" manager="true"/>
      <employees sdo:ref="E0003"/>
    </departments>
  </changeSummary>

  <company:company name="MegaCorp" employeeOfTheMonth="E0004">
    <departments name="Advanced Technologies" location="NY" number="123">
      <employees name="John Jones" SN="E0001"/>
      <employees name="Jane Doe" SN="E0003"/>
      <employees name="Al Smith" SN="E0004" manager="true"/>
    </departments>
  </company:company>
```

```
</sdo:datagraph>
```

The labels above are IDREFs when IDs are available, and SDO XPath expressions otherwise, to locate the data object.

The content of a ChangeSummary element is a deep copy of the objects at the point they were deleted, where the deleted property value was a data object type. The deep copy uses the copy algorithm of the CopyHelper.

Where changes made were only to data type properties, the ChangeSummary element contains copy of the data object from the datagraph, but containing only the properties which have changed, and showing their old values. For example, changing the company name places just the changed information in the change summary.

```
<sdo:datagraph xmlns:company="company.xsd"
               xmlns:sdo="commonj.sdo">

  <changeSummary>
    <company sdo:ref="#/company" name="ACME"/>
  </changeSummary>

  <company:company name="MegaCorp" employeeOfTheMonth="E0004">
    ...
  </company:company>

</sdo:datagraph>
```

If an old value is not present in the ChangeSummary, it is assumed not to have changed. If the old value was not set, the old value is still represented in the ChangeSummary. The XML for old values of datatype properties doesn't distinguish between not present and default value. For example, if comment is an optional property of product and is set for the first time.

```
<sdo:datagraph xmlns:product="product.xsd"
               xmlns:sdo="commonj.sdo">

  <changeSummary>
    <product sdo:ref="#/product">
      <comment/>
    </product>
  </changeSummary>

  <product:product pid="P123">
    <comment>Sale until the end of the month.</comment>
    ...
  </product:product>

</sdo:datagraph>
```

Multi-valued datatype properties and mutli-valued non-cotnainment properties have their entire old and new values in the changeSummary and final XML respectively. For example, if availableColors is a multi-valued property for a product, and the availableColors change:

```
<sdo:datagraph xmlns:product="product.xsd"
               xmlns:sdo="commonj.sdo">

  <changeSummary>
    <product sdo:ref="#/product">
      <availableColors>blue</availableColors>
      <availableColors>green</availableColors>
    </product>
  </changeSummary>

  <product:product pid="P123">
    <availableColors>blue</availableColors>
    <availableColors>red</availableColors>
    ...
  </product:product>

</sdo:datagraph>
```

Examples

The examples given here assume the use of an XML Data Access Service (XMLDAS) to load and save a data graph from and to XML files. The XMLDAS is referenced here to provide a concrete way of illustrating the objects in the graph and to show the effects of operations on the graph in a standard, easily understood format. The code shown here would work just as well against an equivalent data graph that was provided by any other DAS.

The examples covered here include:

The example model is a Company with a Department containing several Employees. The XSD for the Company is shown in the Appendix, "Complete Data Graph for Company Example" on page 140.

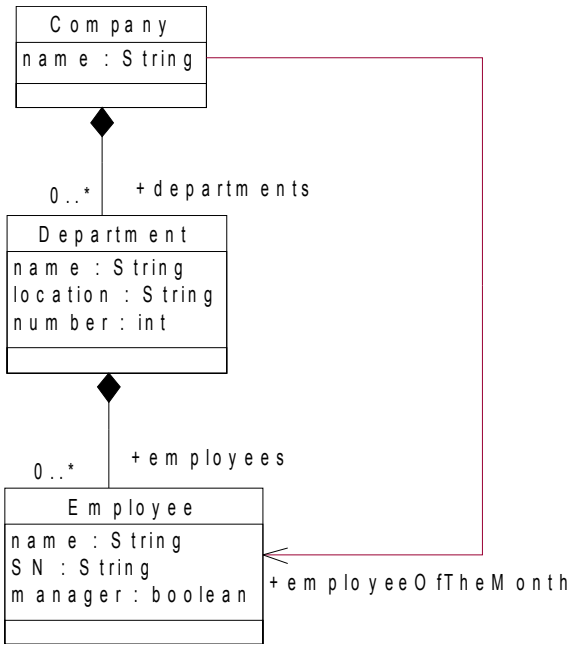


Figure 5: Data Model for Company

Accessing DataObjects using XPath

We can use the XMLHelper to load DataObjects representing a company in a data graph from the following XML file (SN is an XML ID):

```
<sdo:datagraph xmlns:company="company.xsd"
               xmlns:sdo="commonj.sdo">
  <company:company name="ACME" employeeOfTheMonth="E0002">
    <departments name="Advanced Technologies" location="NY" number="123">
      <employees name="John Jones" SN="E0001"/>
      <employees name="Mary Smith" SN="E0002" manager="true"/>
      <employees name="Jane Doe" SN="E0003"/>
    </departments>
  </company:company>
</sdo:datagraph>
```

(This XML conforms to the company model defined in "Complete Data Graph for Company Example" on page 140.)

The examples show how to use DataObjects and the XMLHelper as well as how to use DataGraph. For DataObjects:

```
// Load and access the company DataObject from
// the "company" property of the data graph.
DataObject datagraph = XMLHelper.INSTANCE.load(stream).getRootObject();
DataObject company = datagraph.getDataObject("company");
```

For DataGraph:

```
// Access the company DataObject from the "company" property of
// the root object.
DataObject rootObject = dataGraph.getRootObject();
DataObject company = rootObject.getDataObject("company");
```

If we wish to change the name of the company DataObject from "ACME" to "MegaCorp", we could use the following:

```
// Set the "name" property for the company
company.setString("name", " MegaCorp");
```

Now, suppose we wish to access the employee whose serial number is “E0002”. If we know that this employee is located at index 1 within the department that is located at index 0 within the company object, one way to do this is by traversing each reference in the data graph and locating each DataObject in the many-valued department property using its index in the list. For example, from the company, we can get a list of departments, from that list we can get the department at index 0, from there we can get a list of employees, and from that list we can get the employee at index 1.

```
// Get the list of departments
List departments = company.getList("departments");
// Get the department at index 0 on the list
DataObject department = (DataObject) departments.get(0);
// Get the list of employees for the department
List employees = department.getList("employees");
// Get the employee at index 1 on the list
DataObject employeeFromList = (DataObject) employees.get(1);
```

Alternatively, we can write a single XPath expression that directly accesses the employee from the root company.

```
// Alternatively, an xpath expression can find objects
// based on positions in lists:
DataObject employeeFromXPath =
       company.getDataObject("departments.0/employees.1");
```

Otherwise, if we don’t know the relative positions of the department and employee DataObjects, but we do know that the value number attribute of the department is “123”, we can write an XPath expression that accesses the employee DataObject using the appropriate values:

```
// Get the same employee using an xpath expression
// starting from the company
DataObject employeeFromXPathByValue = company.getDataObject(
   "departments[number=123]/employees[SN='E0002']");
```

In order to remove that employee from the data graph, we could use:

```
// remove the employee from the graph
employeeFromList.detach();
```

And, finally, to create a new employee:

```
// create a new employee
DataObject newEmployee =
            department.createDataObject("employees");
newEmployee.set("name", "Al Smith");
```

```
  newEmployee.set("SN", "E0004");
  newEmployee.setBoolean("manager", true);

  // Reset employeeOfTheMonth to be the new employee
  company.set("employeeOfTheMonth", newEmployee);
```

After saving with the XMLHelper, the resulting XML file would contain:

```
XMLHelper.INSTANCE.save(datagraph, "commonj.sdo", "datagraph", stream);

<sdo:datagraph xmlns:company="company.xsd"
               xmlns:sdo="commonj.sdo">

  <changeSummary create="E0004" delete="E0002">
    <company sdo:ref="#/company" name="ACME" employeeOfTheMonth="E0002"/>
    <departments sdo:ref="#/company/departments[1]">
      <employees sdo:ref="E0001"/>
      <employees name="Mary Smith" SN="E0002" manager="true"/>
      <employees sdo:ref="E0003"/>
    </departments>
  </changeSummary>

  <company:company name="MegaCorp" employeeOfTheMonth="E0004">
    <departments name="Advanced Technologies" location="NY" number="123">
      <employees name="John Jones" SN="E0001"/>
      <employees name="Jane Doe" SN="E0003"/>
      <employees name="Al Smith" SN="E0004" manager="true"/>
    </departments>
  </company:company>

</sdo:datagraph>
```

The ChangeSummary provides an overview of the changes that have been made to the data graph. The ChangeSummary contains DataObjects as they appeared prior to any modifications and includes only those objects and properties that have been modified or deleted or which are referenced by a property that was modified. The sdo:ref attribute is used to map DataObjects, in the ChangeSummary, back to the corresponding DataObjects in the data graph.

In this example, the name property of the Company object was changed, so the original company name is shown in the ChangeSummary. However, the name of the Department object was not changed and therefore the department name does not appear. The employees property of the Department object did change (one Employee was added and one Employee was deleted) so the summary includes the list of all the original employees. In the case of the Employee that was

deleted, all the properties are displayed in the summary. Employees that have not changed include the sdo:ref attribute, but the unchanged properties of these employees are not displayed.

All of the DataObjects in this particular example have been affected or referenced by some change, so the ChangeSummary includes references to all of the objects in the original DataGraph. In another situation where only a few DataObjects from a large data graph are modified, the ChangeSummary would include only small subset of the overall data graph.

Note: The serialized data graph can also have optional elements that describe the model and change information. These elements have been omitted in the output shown above. The complete serialization of this data graph is shown in "Complete Data Graph for Company Example" on page 140.

Accessing DataObjects via Property Index

In the previous section, all the fields in a DataObject were specified using XPath strings, where each string was derived from the name of a property. It is also possible to access fields using the index of each property.

The following example has the same effect as the previous example. The indices for the properties are represented as **int** fields. The values are derived from the position of properties as defined in the company.

```
 // Predefine the property indices
int ROOT_COMPANY = 0;

int COMPANY_DEPARTMENT = 0;
int COMPANY_NAME = 1;

int DEPARTMENT_EMPLOYEES = 0;

int EMPLOYEE_NAME = 0;
int EMPLOYEE_SN = 1;
int EMPLOYEE_MANAGER = 2;

 // Load and access the company DataObject from
 // the "company" property of the data graph.
 DataObject datagraph =XMLHelper.INSTANCE.load(stream).getRootObject();
 DataObject company = datagraph.getDataObject("company");

// Set the "name" property for the company
company.setString(COMPANY_NAME, "MegaCorp");

// Get the list of departments
```

```
List departments = company.getList(COMPANY_DEPARTMENT);
// Get the department at index 0 on the list
DataObject department = (DataObject) departments.get(0);
// Get the list of employees for the department
List employees = department.getList(DEPARTMENT_EMPLOYEES);
// Get the employee at index 1 on the list
DataObject employeeFromList = (DataObject) employees.get(1);

// remove the employee from the graph
employeeFromList.detach();

// create a new employee
DataObject newEmployee =
               department.createDataObject(DEPARTMENT_EMPLOYEES);
newEmployee.set(EMPLOYEE_NAME, "Al Smith");
newEmployee.set(EMPLOYEE_SN, "E0004");
newEmployee.setBoolean(EMPLOYEE_MANAGER, true);
```

Accessing the Contents of a Sequence

The following code uses the Sequence interface to analyze the contents of a data graph that conforms to the Letter model. (The definition of this model is shown in the appendix.) This code first goes through the Sequence looking for unformatted text entries and prints them out. Then the code checks to verify that the contents of the "lastName" property of the DataObject matches the contents of the same property of the Sequence:

```
public static void printSequence(DataObject letter)
{
  // Access the Sequence of the FormLetter
  Sequence letterSequence = letter.getSequence();
  // Print out all the settings that contain unstructured text
  System.out.println("Unstructured text:");
  for (int i=0; i<letterSequence.size(); i++)
  {
    String propertyName = letterSequence.getPropertyName(i);
    if (propertyName=="text")
    {
      String text = (String) letterSequence.getValue(i);
      System.out.println("\t("+text+")");
    }
  }

  // Verify that the lastName property of the DataObject has the same
  // value as the lastName property for the Sequence.
  String dataObjectLastName = letterDataObject.getString("lastName");
  for (int i=0; i<letterSequence.size(); i++)
  {
    String propertyName = letterSequence.getPropertyName(i);
```

```
    if ("lastName".equals(propertyName))
    {
      String sequenceLastName = (String)letterSequence.getValue(i);
      if (dataObjectLastName == sequenceLastName)
        System.out.println("Last Name property matches");
      break;
    }
  }
}
```

Assume that the following XML file is loaded by the XMLDAS to produce a DataGraph that is passed to the above method:

```
  <letter:letters xmlns:letter="http://letterSchema">
    <date>August 1, 2003</date>
    Mutual of Omaha
    Wild Kingdom, USA
    Dear
    <firstName>Casy</firstName>
    <lastName>Crocodile</lastName>
    Please buy more shark repellent.
    Your premium is past due.
  </letter:letters>
```

(Note: this XML conforms to the schema defined in "Complete Data Graph for Letter Example" on page 143.)

The output of this method would be:

```
Unstructured text:
  (Mutual of Omaha)
  (Wild Kingdom, USA)
  (Dear)
  (Please buy more shark repellent.)
  (Your premium is past due.)
Last Name property matches
```

Serializing/Deserializing a DataGraph or DataObject

The DataObject and DataGraph interfaces extend java.io.Serializable, so any DataObject and DataGraph can be serialized. For example, the following code can be used to serialize a given DataObject into a file with a given name:

```
public void serializeDO(DataObject DataObject, String fileName) throws
IOException
  {
```

```
      // serialize data object
      FileOutputStream fos = new FileOutputStream(fileName);
      ObjectOutputStream out = new ObjectOutputStream(fos);
      out.writeObject(DataObject);
      out.close();
  }
```

The following code can be used to deserialize a DataObject from a file with a given name:

```
  public DataObject deserializeDO(String fileName) throws IOException,
ClassNotFoundException
  {
      // de-serialize
      FileInputStream fis = new FileInputStream(fileName);
      ObjectInputStream input = new ObjectInputStream(fis);
      DataObject DataObject = (DataObject) input.readObject();
      input.close();
      return DataObject;
  }
```

Similarly, the following code can be used to serialize and deserialize a DataGraph:

```
  public void serializeDG(DataGraph dataGraph, String fileName) throws
IOException
  {
      // serialize data graph
      FileOutputStream fos = new FileOutputStream(fileName);
      ObjectOutputStream out = new ObjectOutputStream(fos);
      out.writeObject(dataGraph);
      out.close();
    }
  }
  public DataGraph deserializeDG(String fileName) throws IOException,
ClassNotFoundException
  {
      //  de-serialize
      FileInputStream fis = new FileInputStream(fileName);
      ObjectInputStream input = new ObjectInputStream(fis);
      DataGraph deserializedDataGraph = (DataGraph) input.readObject();
      input.close();
      return deserializedDataGraph;
  }
```

Using Type and Property with DataObjects

The Type interface provides access to the metadata for DataObjects in a data graph.

The methods on Type and Property provide information that describes the properties of a DataObject in the data graph. To obtain the Type for a DataObject, use the getType() method.

For example, consider the printDataObject method shown below. This method prints out the contents of a DataObject. Each property is displayed metadata, accessed dynamically, using Type and Property.

```
public void printDataObject(DataObject dataObject, int indent)
{
  // For each Property
  List properties = dataObject.getInstanceProperties();
  for (int p=0, size=properties.size(); p < size; p++)
  {
    if (dataObject.isSet(p))
    {
      Property property = (Property) properties.get(p);
      if (property.isMany())
      {
        // For many-valued properties, process a list of values
        List values = dataObject.getList(p);
        for (int v=0, count=values.size(); v < count; v++)
        {
          printValue(values.get(v), property, indent);
        }
      }
      else
      {
        // For single-valued properties, print out the value
        printValue(dataObject.get(p), property, indent);
      }
    }
  }
}

void printValue(Object value, Property property, int indent)
{
  // Get the name of the property
  String propertyName = property.getName();

  // Construct a string for the proper indentation
  String margin = "";
  for (int i = 0; i < indent; i++)
    margin += "\t";

  if (value != null && property.isContainment())
  {
    // For containment properties, display the value
```

```
    // with printDataObject
    Type type = property.getType();
    String typeName = type.getName();
    System.out.println(margin + propertyName + " (" +typeName+ "):");
    printDataObject((DataObject) value, indent + 1);
   }
   else
   {
    // For non-containment properties, just print the value
    System.out.println(margin + propertyName + ": " + value);
   }
  }
```

For example, consider the following XML file:

```
<sdo:datagraph xmlns:company="company.xsd"
               xmlns:sdo="commonj.sdo">
  <company:company name="ACME" employeeOfTheMonth="E0002">
    <departments name="Advanced Technologies" location="NY" number="123">
      <employees name="John Jones" SN="E0001"/>
      <employees name="Mary Smith" SN="E0002" manager="true"/>
      <employees name="Jane Doe" SN="E0003"/>
    </departments>
  </company:company>
</sdo:datagraph>
```

(Note: this XML conforms to the company model XSD defined in "Complete Data Graph for Company Example" on page 140.)

If this file is loaded using an XML Data Mediator Service, the resulting data graph could be printed out using:

```
   printDataObject(dataGraph.getRootObject(), 0);
```

The console output for this data graph would be:

```
company (Company):
 name: ACME
 departments (Department):
  name: Advanced Technologies
  location: NY
  number: 123
  employees (Employee):
   name: John Jones
   SN: E0001
  employees (Employee):
```

```
    name: Mary Smith
    SN: E0002
    manager: true
  employees (Employee):
    name: Jane Doe
    SN: E0003
employeeOfTheMonth: Employee (name=Mary Smith, SN=E0002,
           manager=true, employeeStatus=fullTime)
```

Creating XML from Data Objects

The following code will create and save a purchase order, as shown in the XSD primer. This example makes use of DataFactory and XMLHelper:

```
DataObject purchaseOrder =
  DataFactory.INSTANCE.create(null, "PurchaseOrderType");

purchaseOrder.setString("orderDate", "1999-10-20");

DataObject shipTo = purchaseOrder.createDataObject("shipTo");
shipTo.set("country", "US");
shipTo.set("name", "Alice Smith");
shipTo.set("street", "123 Maple Street");
shipTo.set("city", "Mill Valley");
shipTo.set("state", "CA");
shipTo.setString("zip", "90952");

DataObject billTo = purchaseOrder.createDataObject("billTo");
billTo.set("country", "US");
billTo.set("name", "Robert Smith");
billTo.set("street", "8 Oak Avenue");
billTo.set("city", "Mill Valley");
shipTo.set("state", "PA");
billTo.setString("zip", "95819");
purchaseOrder.set("comment", "Hurry, my lawn is going wild!");
DataObject items = purchaseOrder.createDataObject("items");

DataObject item1 = items.createDataObject("item");
item1.set("partNum", "872-AA");
item1.set("productName", "Lawnmower");
item1.setInt("quantity", 1);
item1.setString("USPrice", "148.95");
item1.set("comment", "Confirm this is electric");

DataObject item2 = items.createDataObject("item");
item2.set("partNum", "926-AA");
```

```
    item2.set("productName", "Baby Monitor");
    item1.setInt("quantity", 1);
    item2.setString("USPrice", "39.98");
    item2.setString("shipDate", "1999-05-21");

    OutputStream stream = new FileOutputStream("myPo.xml");
    XMLHelper.INSTANCE.save(purchaseOrder, null, "purchaseOrder",      stream);
```

The following output is created:

```
<?xml version="1.0" encoding="UTF-8"?>
<purchaseOrder orderDate="1999-10-20">
  <shipTo country="US">
    <name>Alice Smith</name>
     <street>123 Maple Street</street>
    <city>Mill Valley</city>
    <state>PA</state>
    <zip>90952</zip>
  </shipTo>
  <billTo country="US">
    <name>Robert Smith</name>
    <street>8 Oak Avenue</street>
    <city>Mill Valley</city>
    <zip>95819</zip>
  </billTo>
  <comment>Hurry, my lawn is going wild!</comment>
  <items>
    <item partNum="872-AA">
      <productName>Lawnmower</productName>
      <quantity>1</quantity>
      <USPrice>148.95</USPrice>
      <comment>Confirm this is electric</comment>
    </item>
    <item partNum="926-AA">
      <productName>Baby Monitor</productName>
      <USPrice>39.98</USPrice>
      <shipDate>1999-05-21</shipDate>
    </item>
  </items>
</purchaseOrder>
```

Creating DataObject Trees from XML documents

It is possible to convert to and from XML documents to build DataObject trees, which is useful when assembling DataObjects from several data sources. XMLHelper can be used to do this. For

example, suppose the global elements for shipTo and billTo were declared in the PurchaseOrder XSD:

```
<element name="shipTo" type="USAddress"/>
<element name="billTo" type="USAddress"/>
```

To create the shipTo DataObject from XML:

```
String shipToXML =
  "<shipTo country='US'"+
  "  <name>Alice Smith</name>"+
  "  <street>123 Maple Street</street>"+
  "  <city>Mill Valley</city>"+
  "  <state>PA</state>"+
  "  <zip>90952</zip>"+
  "</shipTo>";
DataObject shipTo = XMLHelper.INSTANCE.load(shipToXML).getRootObject();
purchaseOrder.set("shipTo", shipTo);
```

To convert the billTo DataObject to XML:

```
String billToXML = XMLHelper.INSTANCE.save(billTo, null, "billTo");
System.out.println(billToXML);
```

This produces:

```
<?xml version="1.0" encoding="UTF-8"?>
<billTo country="US"
  <name>Robert Smith</name>
  <street>8 Oak Avenue</street>
  <city>Mill Valley</city>
  <zip>95819</zip>
</billTo>
```

Creating open content XML documents

Open content is often used when a DataObject allows new Properties to be used even when they are not known in advance. This occurs often in XML, for example in Web Services where a SOAP envelope is used to wrap contents specific to web service invocations. In the case of SOAP, an Envelope element contains a Body element and the Body element has open content to allow any element inside. This example shows how to make DataObjects for the SOAP Envelope and Body and place inside a Purchase Order.

```
// Create a SOAP envelope and body
```

```
String soap = "http://schemas.xmlsoap.org/wsdl/soap/";
DataObject envelope = DataFactory.INSTANCE.create(soap, "Envelope");
DataObject body = envelope.createDataObject("Body");

// The Body is open content.
// Create a purchase order using the XML global element purchaseOrder
Property poProperty = XSDHelper.INSTANCE.getGlobalProperty(null,
                                                "purchaseOrder", true);
DataObject po = body.createDataObject(poProperty);

// fill out the rest of the purchase order
po.set("orderDate", "2005-06-10");
// ...
```

If the purchase order already existed, instead of calling body.create(), call body.set().

```
body.set(poProperty, existingPo);
```

Using the purchase order in a web service and getting the results is straightforward, by invoking the web service and then extracting from the return soap envelope the result purchase order.

```
DataObject resultEnvelope = WebService.invoke(
   po, "http://werbservices.org/purchaseOrder", soap, "Envelope");

// Get the purchase order from the result envelope
DataObject resultPo =
   resultEnvelope.getDataObject("Body/purchaseOrder");
```

Web Services Client using XMLHelper

A simple web services client can be built around the XMLHelper. In this web service client, an input DataObject representing an XML document is POSTed using the XMLHelper, and the returning XML document is returned to the caller as a DataObject. More advanced web service clients would be interested in the SOAP header.

```
public static DataObject invoke(DataObject input, String serviceUri,
    String rootElementURI, String rootElementName)  throws IOException
{
  URL address = new URL(serviceUri);
  HttpURLConnection connection = (HttpURLConnection)
    address.openConnection();

  if (input != null)
  {
    connection.setRequestMethod("POST");
    connection.setDoOutput(true);
    connection.addRequestProperty("Content-Type",
```

```
      "text/xml; charset=utf-8");
    OutputStream os = connection.getOutputStream();
    // Add the XML document to the request
    XMLHelper.INSTANCE.save(input, rootElementURI, rootElementName, os);
    os.flush();
  }

  // invoke the service
  connection.connect();
  int code = connection.getResponseCode();
  if (code != HttpURLConnection.HTTP_OK)
  {
    throw new IOException("HTTP "+code+" "+
        connection.getResponseMessage());
  }

  InputStream is = connection.getInputStream();
  // Return the root DataObject from the web service response
  DataObject output = XMLHelper.INSTANCE.load(is).getRootObject();
  return output;
}
```

Web services and DataGraphs Example

Data graphs may be used in Web services by passing the <datagraph> element in the body of a soap message. For example, the data graph in these examples could be included in a soap body sent on the wire in a web service invocation.

```
<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/">
  <soap:Header/>
  <soap:Body>
    <sdo:datagraph
      xmlns:company="company.xsd"
      xmlns:sdo="commonj.sdo">
      <company:company name="ACME" employeeOfTheMonth="E0002">
        <departments name="Advanced Technologies" location="NY" number="123">
          <employees name="John Jones" SN="E0001"/>
          <employees name="Mary Smith" SN="E0002" manager="true"/>
          <employees name="Jane Doe" SN="E0003"/>
        </departments>
      </company:company>
    </sdo:datagraph>
  </soap:Body>
</soap:Envelope>
```

The SDO BaseDataGraphType allows any root DataObject to be included with the “any” element declaration. To constrain the type of root DataObject in DataGraph XML, an extended DataGraph, CompanyDataGraph, can be declared that restricts the type to a single expected kind, CompanyType. The XSD declaration is from the appendix “Complete Data Graph for Company Example” on page 140.

```
<xsd:element name="company" type="company:CompanyType"/>
<xsd:complexType name="CompanyType">
  <xsd:sequence>
    <xsd:element name="departments" type="company:DepartmentType"
maxOccurs="unbounded"/>
  </xsd:sequence>
  <xsd:attribute name="name" type="xsd:string"/>
  <xsd:attribute name="employeeOfTheMonth" type="xsd:string"/>
</xsd:complexType>
```

This example shows a companyDataGraph with a CompanyType root DataObject. These XSD declarations define a CompanyDataGraph extending SDO BaseDataGraphType with CompanyType as the type of root DataObject instead of any.

```
<element name="companyDatagraph" type="company:CompanyDataGraphType"/>
<complexType name="CompanyDataGraphType">
  <complexContent>
    <extension base="sdo:BaseDataGraphType">
      <sequence>
        <element name="company" type="company:CompanyType"/>
      </sequence>
    </extension>
  </complexContent>
</complexType>
```

This ensures that only the company element may appear as the root DataObject of the data graph. The SOAP message for the companyDatagraph is:

```
<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/">
 <soap:Header/>
 <soap:Body>
    <company:companyDatagraph
      xmlns:company="company.xsd">
      <company:company name="ACME" employeeOfTheMonth="E0002">
        <departments name="Advanced Technologies" location="NY" number="123">
          <employees name="John Jones" SN="E0001"/>
          <employees name="Mary Smith" SN="E0002" manager="true"/>
          <employees name="Jane Doe" SN="E0003"/>
        </departments>
```

```
      </company:company>
    </company:companyDatagraph>
  </soap:Body>
</soap:Envelope>
```

The WSDL for the Web service with the companyDatagraph is below. The full listing is shown in the appendix in "Complete WSDL for Web services Example" on page 143.

```
<wsdl:definitions name="Name"
  targetNamespace="http://example.com"
  xmlns:tns="http://example.com"
  xmlns:company="company.xsd"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <wsdl:types>
    <schema xmlns="http://www.w3.org/2001/XMLSchema"
      targetNamespace="company.xsd"
      xmlns:company="company.xsd"
      xmlns:sdo="commonj.sdo"
      elementFormDefault="qualified">
        <element name="companyDatagraph" type="company:CompanyDataGraphType"/>
        <complexType name="CompanyDataGraphType">
          <complexContent>
            <extension base="sdo:BaseDataGraphType">
              <sequence>
                <element name="company" type="company:CompanyType"/>
              </sequence>
            </extension>
          </complexContent>
        </complexType>
    </schema>
  </wsdl:types>
  ...
</wsdl:definitions>
```

Complete Data Graph Examples

Complete Data Graph Serialization

As mentioned in the section on Data Graph Serialization, the serialization of a data graph includes optional elements, that describe the model and the change information, in addition to the DataObjects in the data graph.

The model may be described either as an instance of an XML Schema or EMOF Package (See "Complete Data Graph for Company Example" on page 140) or using an XML Schema (see "Complete Data Graph for Letter Example" on page 143.)

Complete Data Graph for Company Example

The following XML represents the complete serialization of the data graph that includes the changes from the processing described in "Accessing DataObjects using XPath" on page 124.

```
<sdo:datagraph xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
               xmlns:xsd="http://www.w3.org/2001/XMLSchema"
               xmlns:company="company.xsd"
               xmlns:sdo="commonj.sdo">
  <xsd>
    <xsd:schema targetNamespace="company.xsd">
      <xsd:element name="company" type="company:CompanyType"/>
      <xsd:complexType name="CompanyType">
        <xsd:sequence>
          <xsd:element name="departments" type="company:DepartmentType"
maxOccurs="unbounded"/>
        </xsd:sequence>
        <xsd:attribute name="name" type="xsd:string"/>
        <xsd:attribute name="employeeOfTheMonth" type="xsd:string"/>
      </xsd:complexType>
      <xsd:complexType name="DepartmentType">
        <xsd:sequence>
          <xsd:element name="employees" type="company:EmployeeType"
maxOccurs="unbounded"/>
        </xsd:sequence>
        <xsd:attribute name="name" type="xsd:string"/>
        <xsd:attribute name="location" type="xsd:string"/>
        <xsd:attribute name="number" type="xsd:int"/>
      </xsd:complexType>
      <xsd:complexType name="EmployeeType">
        <xsd:attribute name="name" type="xsd:string"/>
        <xsd:attribute name="SN" type="xsd:ID"/>
        <xsd:attribute name="manager" type="xsd:boolean"/>
```

```
      </xsd:complexType>
    </xsd:schema>
  </xsd>
  <changeSummary create="E0004" delete="E0002">
    <company sdo:ref="#/company" name="ACME" employeeOfTheMonth="E0002"/>
    <departments sdo:ref="#/company/departments[1]">
      <employees sdo:ref="E0001"/>
      <employees name="Mary Smith" SN="E0002" manager="true"/>
      <employees sdo:ref="E0003"/>
    </departments>
  </changeSummary>

  <company:company name="MegaCorp" employeeOfTheMonth="E0004">
    <departments name="Advanced Technologies" location="NY" number="123">
      <employees name="John Jones" SN="E0001"/>
      <employees name="Jane Doe" SN="E0003"/>
      <employees name="Al Smith" SN="E0004" manager="true"/>
    </departments>
  </company:company>
</sdo:datagraph>
```

When using EMOF as metadata, the complete data graph serialization is:

```
<sdo:datagraph xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
               xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
               xmlns:company="company.xsd"
               xmlns:emof="http://schema.omg.org/spec/mof/2.0/emof.xmi"
               xmlns:sdo="commonj.sdo">
  <models>
    <emof:Package name="companyPackage"
                  uri="companySchema.emof">
      <ownedType xsi:type="emof:Class" name="CompanySchema">
        <ownedProperty name="company" type="#model.0" containment="true"/>
      </ownedType>
      <ownedType xsi:type="emof:Class" xmi:id="model.0" name="Company">
        <ownedProperty name="departments" type="#model.1" upperBound="-1"
                       containment="true"/>
        <ownedProperty name="employeeOfTheMonth" type="#model.7"/>
        <ownedProperty name="name">
              <type xsi:type="emof:DataType"
                  href="http://schema.omg.org/spec/mof/2.0/emof.xmi#String"/>
        </ownedProperty>
      </ownedType>
      <ownedType xsi:type="emof:Class" xmi:id="model.1" name="Department">
        <ownedProperty name="employees" type="#model.2" upperBound="-1"
                       containment="true"/>
        <ownedProperty name="name">
              <type xsi:type="emof:DataType"
```

```
                href="http://schema.omg.org/spec/mof/2.0/emof.xmi#String"/>
        </ownedProperty>
        <ownedProperty name="location" >
              <type xsi:type="emof:DataType"
                  href="http://schema.omg.org/spec/mof/2.0/emof.xmi#String"/>
        </ownedProperty>
        <ownedProperty name="number" >
              <type xsi:type="emof:DataType"
                  href="http://schema.omg.org/spec/mof/2.0/emof.xmi#Integer"/>
        </ownedProperty>
      </ownedType>
      <ownedType xsi:type="emof:Class" xmi:id="model.2" name="Employee">
        <ownedProperty name="name">
              <type xsi:type="emof:DataType"
                  href="http://schema.omg.org/spec/mof/2.0/emof.xmi#String"/>
        </ownedProperty>
        <ownedProperty name="SN">
              <type xsi:type="emof:DataType"
                  href="http://schema.omg.org/spec/mof/2.0/emof.xmi#String"/>
        </ownedProperty>
        <ownedProperty name="manager">
              <type xsi:type="emof:DataType"
                  href="http://schema.omg.org/spec/mof/2.0/emof.xmi#Boolean"/>
        </ownedProperty>
        <ownedProperty name="employeeStatus" type="#model.3"/>
      </ownedType>
      <ownedType xsi:type="emof:Enumeration" xmi:id="model.3">
        <ownedLiteral name="fullTime" value="1"/>
        <ownedLiteral name="partTime" value="2"/>
      </ownedType>
    </emof:Package>
  </models>
  <changeSummary create="#id.4" delete="#log.0">
    <company sdo:ref="#id.0" name="ACME" employeeOfTheMonth="#log.0"/>
    <departments sdo:ref="#id.1">
      <employees sdo:ref="#id.2"/>
      <employees xmi:id="log.0" name="Mary Smith" SN="E0002" manager="true"/>
      <employees sdo:ref="#id.3"/>
    </departments>
  </changeSummary>
  <company:company xmi:id="id.0" name="MegaCorp" employeeOfTheMonth="#id.4">
    <departments xmi:id="id.1" name="Advanced Technologies" location="NY"
number="123">
      <employees xmi:id="id.2" name="John Jones" SN="E0001"/>
      <employees xmi:id="id.3" name="Jane Doe" SN="E0003"/>
      <employees xmi:id="id.4" name="Al Smith" SN="E0004" manager="true"/>
    </departments>
  </company:company>
</sdo:datagraph>
```

Complete Data Graph for Letter Example

This data graph is used as the input for the example shown in "Accessing the Contents of a Sequence " on page 128. In this case the XSD for the letter is sent as an option, along with the DataObjects. No summary information is sent. When the receiver reads the data graph, the XSD is the metadata and the letter is the data.

```
<sdo:datagraph xmlns:sdo="commonj.sdo"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:letter="http://letterSchema">
  <xsd>
    <xsd:schema targetNamespace="letter.xsd">
      <xsd:element name="letters" type="letter:FormLetter"/>
      <xsd:complexType name="FormLetter" mixed="true">
        <xsd:sequence>
          <xsd:element name="date" minOccurs="0" type="xsd:string"/>
          <xsd:element name="firstName" minOccurs="0" type="xsd:string"/>
          <xsd:element name="lastName" minOccurs="0" type="xsd:string"/>
        </xsd:sequence>
      </xsd:complexType>
    </xsd:schema>
  </xsd>
  <letter:letters>
    <date>August 1, 2003</date>
    Mutual of Omaha
    Wild Kingdom, USA
    Dear
    <firstName>Casy</firstName>
    <lastName>Crocodile</lastName>
    Please buy more shark repellent.
    Your premium is past due.
  </letter:letters>
</sdo:datagraph>
```

Complete WSDL for Web services Example

The full WSDL from the Using Web services with data graph Example.

```
<wsdl:definitions name="Name"
  targetNamespace="http://example.com"
  xmlns:tns="http://example.com"
  xmlns:company="company.xsd"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
```

```
  <wsdl:types>
   <schema xmlns="http://www.w3.org/2001/XMLSchema"
     targetNamespace="company.xsd"
     xmlns:company="company.xsd"
     xmlns:sdo="commonj.sdo"
     elementFormDefault="qualified">
       <element name="companyDatagraph" type="company:CompanyDataGraphType"/>
       <complexType name="CompanyDataGraphType">
         <complexContent>
           <extension base="sdo:BaseDataGraphType">
             <sequence>
               <element name="company" type="company:CompanyType"/>
             </sequence>
           </extension>
         </complexContent>
       </complexType>
   </schema>
  </wsdl:types>
  <wsdl:message name="fooMessage">
    <wsdl:part name="body" element="company:companyDataGraph"/>
  </wsdl:message>
  <wsdl:message name="fooResponseMessage"></wsdl:message>
  <wsdl:portType name="fooPortType">
    <wsdl:operation name="myOperation">
     <wsdl:input message="tns:fooMessage"/>
     <wsdl:output message="tns:fooResponseMessage"/>
    </wsdl:operation>
  </wsdl:portType>
  <wsdl:binding name="fooBinding" type="tns:fooPortType">
    <soap:binding style="document"
     transport="http://schemas.xmlsoap.org/soap/http"/>
    <wsdl:operation name="myOperation">
     <soap:operation/>
     <wsdl:input>
      <soap:body use="literal"/>
     </wsdl:input>
     <wsdl:output>
      <soap:body use="literal"/>
     </wsdl:output>
    </wsdl:operation>
  </wsdl:binding>
  <wsdl:service name="myService">
    <wsdl:port name="myPort" binding="tns:fooBinding">
     <soap:address location="http://localhost/myservice"/>
    </wsdl:port>
  </wsdl:service>
</wsdl:definitions>
```

DataType Conversions

Conversions with an x are supported through DataObject or DataHelper. X are identity where the input and output are the same. Other conversions are not supported, including combinations not in these tables. Conversions between primitives and object representations are supported by the Java language and through DataObject. Conversions between the primitive and object wrapper form are also supported. Conversions of Lists to String are as specified by java.util.AbstractCollection.toString().

To-> From | V	Boolean	Byte	Character	Double	Float	Int	Long	Short	String	Bytes	Decimal	Integer	Date
Boolean	X								x				
Byte		X		x	x	x	x	x	x				
Character			X						x				
Double		x		X	x	x	x	x	x		x	x	
Float		x		x	X	x	x	x	x		x	x	
Int		x		x	x	X	x	x	x		x	x	
Long		x		x	x	x	X	x	x		x	x	x
Short		x		x	x	x	x	X	x				
String	x	x	x	x	x	x	x	x	X		x	x	x
Bytes										X		x	
Decimal				x	x	x	x		x		X	x	
Integer				x	x	x	x		x	x	x	X	
Date							x		x				X

To-> From | V	String	Day	Date	DateTime	Duration	Month	MonthDay	Strings	Time	Year	YearMonth	YearMonthDay
String	X	x	x	x	x	x	x	x	x	x	x	x
Day	x	X	x									
Date	x	x	X	x	x	x	x		x	x	x	x
DateTime	x		x	X								
Duration	x		x		X							
Month	x		x			X						
MonthDay	x		x				X					
Strings	x							X				
Time	x		x						X			
Year	x		x							X		
YearMonth	x		x								X	
YearMonthDay	x		x									X

Acknowledgements

We would like to thank Johsua Auerbach (IBM), David Bau (BEA), David Booz (IBM), Adam Bosworth (BEA), Graham Barber (IBM), Kevin Bauer (IBM), Michael Beisiegel (IBM), Frank Budinsky (IBM), Graham Charters (IBM), Colin Thorne (IBM), Shane Claussen (IBM), Ed Cobb (BEA), Brent Daniel (IBM), George DeCandio (IBM), Jean-Sebastien Delfino (IBM), Scott Dietzen (BEA), Mike Edwards (IBM), Emma Eldergill (IBM), Don Ferguson (IBM), Christopher Ferris (IBM), Paul Fremantle (IBM), Kelvin Goodson (IBM), John Green (IBM), Laurent Hasson (IBM), Eric Herness (IBM), Rob High (IBM), Steve Holbrook (IBM), Sridhar Iyengar (IBM), Jagan Karuturi (IBM), Stephen J Kinder (IBM), Elena Litani (IBM), Matthew Lovett (IBM), Angel Luis Diaz (IBM), Ed Merks (IBM), Adam Messinger (BEA), Simon Nash (IBM), Peter Niblett (IBM), Karla Norsworthy (IBM), Howard Operowsky (IBM), Bertrand Portier (IBM), Barbara Price (IBM), Jim Rhyne (IBM), Fabio Riccardi (BEA), Timo Salo (IBM), Edward Slattery (IBM), Dave Steinberg (IBM), Andrew Spyker (IBM), Greg Truty (IBM), Celia Tung (IBM), Seth White (BEA), Kevin Williams (IBM), and George Zagelow (IBM).

Trademarks

References

[1] EMOF compliance point from Meta Object Facility 2.0 Core Final Submission,
http://www.omg.org/cgi-bin/doc?ad/2003-04-07

[2] XML Schema Part 1: Structures,
http://www.w3.org/TR/xmlschema-1

[3] Next-Generation Data Programming with Service Data Objects
Any one of:

- http://dev2dev.bea.com/technologies/commonj/index.jsp
- http://www.ibm.com/developerworks/library/specification/ws-sdo/
- http://oracle.com/technology/webservices/sca
- https://www.sdn.sap.com/
- http://www.xcalia/xdn/specs/sdo
- http:/www.sybase.com

[4] MOF2 XMI Final submission
http://www.omg.org/docs/ad/03-04-04.pdf

[5] XPath 1.0 specification
http://www.w3.org/TR/xpath

[6] Java 1.5.0 API documentation
http://java.sun.com/j2se/1.5.0/

[7] XML Schema Part 2: Datatypes
http://www.w3.org/TR/xmlschema-2