

Feature Data Objects (FDO)

Developer's Guide

The Autodesk logo is displayed in white text on a black rectangular background. The word "Autodesk" is written in a sans-serif font, with a registered trademark symbol (®) at the end. The logo is oriented vertically, reading from bottom to top.

Autodesk®

March 2008

© 2008 Autodesk, Inc. All Rights Reserved. Except as otherwise permitted by Autodesk, Inc., this publication, or parts thereof, may not be reproduced in any form, by any method, for any purpose.

Certain materials included in this publication are reprinted with the permission of the copyright holder.

Trademarks

The following are registered trademarks or trademarks of Autodesk, Inc., in the USA and other countries: 3DEC (design/logo), 3December, 3December.com, 3ds Max, ActiveShapes, Actrix, ADI, Alias, Alias (swirl design/logo), AliasStudio, AliasWavefront (design/logo), ATC, AUGI, AutoCAD, AutoCAD Learning Assistance, AutoCAD LT, AutoCAD Simulator, AutoCAD SQL Extension, AutoCAD SQL Interface, Autodesk, Autodesk Envision, Autodesk Insight, Autodesk Intent, Autodesk Inventor, Autodesk Map, Autodesk MapGuide, Autodesk Streamline, AutoLISP, AutoSnap, AutoSketch, AutoTrack, Backdraft, Built with ObjectARX (logo), Burn, Buzzsaw, CAiCE, Can You Imagine, Character Studio, Cinestream, Civil 3D, Cleaner, Cleaner Central, ClearScale, Colour Warper, Combustion, Communication Specification, Constructware, Content Explorer, Create>what's>Next> (design/logo), Dancing Baby (image), DesignCenter, Design Doctor, Designer's Toolkit, DesignKids, DesignProf, DesignServer, DesignStudio, DesignStudio (design/logo), Design Your World, Design Your World (design/logo), DWF, DWG, DWG (logo), DWG TrueConvert, DWG TrueView, DXF, EditDV, Education by Design, Exposure, Extending the Design Team, FBX, Filmbox, FMDesktop, Freewheel, GDX Driver, Gmax, Heads-up Design, Heidi, HOOPS, HumanIK, i-drop, iMOUT, Incinerator, IntroDV, Inventor, Inventor LT, Kaydara, Kaydara (design/logo), LocationLogic, Lustre, Maya, Mechanical Desktop, MotionBuilder, Mudbox, NavisWorks, ObjectARX, ObjectDBX, Open Reality, Opticore, Opticore Opus, PolarSnap, PortfolioWall, Powered with Autodesk Technology, Productstream, ProjectPoint, ProMaterials, Reactor, RealDWG, Real-time Roto, Recognize, Render Queue, Reveal, Revit, Showcase, ShowMotion, SketchBook, SteeringWheels, StudioTools, Topobase, Toxik, ViewCube, Visual, Visual Bridge, Visual Construction, Visual Drainage, Visual Hydro, Visual Landscape, Visual Roads, Visual Survey, Visual Syllabus, Visual Toolbox, Visual Tugboat, Visual LISP, Voice Reality, Volo, Wiretap, and WiretapCentral

The following are registered trademarks or trademarks of Autodesk Canada Co. in the USA and/or Canada and other countries: Backburner, Discreet, Fire, Flame, Flint, Frost, Inferno, Multi-Master Editing, River, Smoke, Sparks, Stone, and Wire

All other brand names, product names or trademarks belong to their respective holders.

Disclaimer

THIS PUBLICATION AND THE INFORMATION CONTAINED HEREIN IS MADE AVAILABLE BY AUTODESK, INC. "AS IS." AUTODESK, INC. DISCLAIMS ALL WARRANTIES, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE REGARDING THESE MATERIALS.

Published By: Autodesk, Inc.
111 McInnis Parkway
San Rafael, CA 94903, USA

Contents

Chapter 1	About This Guide	1
	Audience and Purpose	1
	How This Guide Is Organized	1
	What's New	3
Chapter 2	Introduction	5
	What Is the FDO API?	5
	From the Perspective of the Client Application User	5
	From the Perspective of the Client Application Engineer	5
	Getting Started	6
	FDO Architecture and Providers	7
	What Is a Provider?	9
	Developing Applications	11
Chapter 3	FDO Concepts	13
	Data Concepts	13
	Operational Concepts	18
Chapter 4	Development Practices	21
	Memory Management	21
	Collections	23
	Exception Handling	23

	Exception Messages	24
	Managing FdoPtr Behaviors	25
Chapter 5	Establishing a Connection	27
	Connection Semantics	27
	Establishing a Connection	29
Chapter 6	Capabilities	35
	What Is the Capabilities API?	35
	Connection Capabilities	36
	Code	36
	Schema Capabilities	37
	Code	37
	Command Capabilities	40
	Code	40
	Expression Capabilities	41
	Code	41
	Filter Capabilities	42
	Code	42
	Geometry Capabilities	43
	Code	43
	Raster Capabilities	44
	Code	44
Chapter 7	Schema Management	47
	Schema Package	47
	Schema Overrides	49
	Working with Schemas	49
	FDOFeatureClass	51
	FDOClass	51
	Non-Feature Class Issues	52
	Modifying Models	55
	Schema Element States	55
	Rollback Mechanism	56
	FDO XML Format	56
	Creating and Editing a GML Schema File	62
	Schema Management Examples	72
Chapter 8	Data Maintenance	79
	Data Maintenance Operations	79
	Inserting Values	79
	Updating Values	85
	Deleting Values	86
	Related Class Topics	87

Chapter 9	Performing Queries	89
	Creating a Query	89
	Query Example	90
Chapter 10	Long Transaction Processing	95
	What Is Long Transaction Processing?	95
	Supported Interfaces	96
Chapter 11	Filter and Expression Languages	99
	Filters	99
	Expressions	100
	Filter and Expression Text	100
	Language Issues	101
	Provider-Specific Constraints on Filter and Expression Text	101
	Filter Grammar	101
	Expression Grammar	103
	Filter and Expression Keywords	104
	Data Types	104
	Identifier	104
	Parameter	104
	String	104
	Integer	105
	Double	105
	DateTime	105
	Operators	106
	Special Character	107
	Geometry Value	107
Chapter 12	The Geometry API	111
	Introduction	111
	FGF and WKB	111
	FGF Binary Specification	112
	FGF Text	118
	Abstract and Concrete Classes	118
	Geometry Types	119
	Mapping Between Geometry and Geometric Types	120
	Spatial Context	120
	Specify Dimensionality When Creating Geometries Using String Specifications	121
	Inserting Geometry Values	122
Appendix A	Autodesk FDO Provider for Oracle	123
	What Is FDO Provider for Oracle?	123

FDO Provider for Oracle General Requirements	124
FDO Provider for Oracle Connection	124
FDO Provider for Oracle and Foreign Schemas	125
Foreign Schema Settings	125
Settings on the FDO Schema Instance	125
Settings on the Foreign Schema Instance	125
Oracle Identity Property	126
Read-Write Privileges	126
Foreign Schema Limitations	126
Ensuring Valid Views When Applying a Feature Schema Against a Foreign Schema	126
Table Name Restrictions When Working with a Foreign Schema	129
Schema Access on a Different Oracle Instance	129
FDO Provider for Oracle and Schema Overrides	130
Schema Override Set	130
Class Table Overrides	131
Data Property Overrides	131
Object Property Overrides	131
Geometric Property Overrides	131
Oracle-Specific Schema Creation Restrictions	132
FDOFeatureClass	132
Classes	132
Properties	132
Data Properties	132
Identity Properties	133
String Properties	133
Decimal Properties	133
Geometric Properties	133
Object Properties	133
Oracle-Specific Schema Modification Restrictions	134
Schema Element Descriptions	134
Data Properties	134
Oracle-Specific Deletion Restrictions	134
FDOClassDefinition	134
FDOClass	135
Property	135
Oracle Reserved Words Used with Filter and Expression Text	135
Locking and Long Transactions	136
OWM and FDO Lock Types	137
Example: AllLongTransactionExclusiveLock	137
FDO Provider for Oracle Capabilities	139
 Appendix B OSGeo FDO Provider for ArcSDE	 147
What Is FDO Provider for ArcSDE?	147
FDO Provider for ArcSDE Software Requirements	147

	Installed Components	147
	External Dependencies	148
	FDO Provider for ArcSDE Limitations	148
	ArcSDE Limitations	148
	Relative to ArcObjects API and ArcGIS Server API	149
	Curved Segments	149
	Locking and Versioning	149
	Table Creation	150
	Identity Row ID Column and Enable Row Locking	150
	Disable Row Locking and Enable Versioning	151
	FDO Provider for ArcSDE Connection	152
	Data Type Mappings	153
	Creating a Feature Schema	154
	FDO Provider for ArcSDE Capabilities	159
Appendix C	OSGeo FDO Provider for MySQL	165
	What Is FDO Provider for MySQL?	165
	FDO Provider for MySQL Capabilities	166
Appendix D	OSGeo FDO Provider for ODBC	173
	What Is FDO Provider for ODBC?	173
	FDO Provider for ODBC Capabilities	174
Appendix E	Autodesk FDO Provider for Raster	179
	What Is FDO Provider for Raster?	179
	FDO Provider for Raster Capabilities	181
Appendix F	OSGeo FDO Provider for SDF	187
	What Is FDO Provider for SDF?	187
	FDO Provider for SDF Capabilities	188
Appendix G	OSGeo FDO Provider for SHP	195
	What Is FDO Provider for SHP?	195
	FDO Provider for SHP Capabilities	196
Appendix H	Autodesk FDO Provider for SQL Server	201
	What Is FDO Provider for SQL Server?	201
	FDO Provider for SQL Server Capabilities	201
Appendix I	OSGeo FDO Provider for WFS	209
	What Is FDO Provider for WFS?	209
	FDO Provider for WFS Capabilities	210

Appendix J	OSGeo FDO Provider for WMS	215
	What Is FDO Provider for WMS?	215
	FDO Provider for WMS Capabilities	216
Appendix K	Expression Functions	221
	Introduction	221
	Expression Function List	221
	Aggregate Expression Functions	221
	Conversion Expression Functions	222
	Date Expression Functions	223
	Geometry Expression Functions	223
	Mathematical Expression Functions	224
	Numeric Expression Functions	225
	String Expression Functions	225
	Index	227

About This Guide



The *FDO Developer's Guide* introduces the Feature Data Objects (FDO) application programming interface (API) and explains how to use its customization and development features.

NOTE For detailed information about installing the FDO SDK and getting started using the FDO API, see *The Essential FDO* (FET_TheEssentialFDO).

Audience and Purpose

This guide is intended to be used by developers of FDO applications. It introduces the FDO API, explains the role of a feature provider, and provides detailed information and examples about how to code your application.

How This Guide Is Organized

This guide consists of the following chapters and appendixes:

- [Introduction](#) on page 5, provides an overview of the FDO API and the function of FDO feature providers.
- [FDO Concepts](#) on page 13, describes the key data and operational concepts upon which FDO is constructed.
- [Development Practices](#) on page 21, discusses the best practices to follow when using FDO for application development.
- [Establishing a Connection](#) on page 27, describes how to establish a connection to an FDO provider.

- [Capabilities](#) on page 35, discusses the Capabilities API, which is used to determine the capabilities of a particular provider.
- [Schema Management](#) on page 47, describes how to create and work with schemas and presents the issues related to schema management.
- [Data Maintenance](#) on page 79, provides information about using the FDO API to maintain the data.
- [Performing Queries](#) on page 89, describes how to create and perform queries.
- [Long Transaction Processing](#) on page 95, discusses long transactions (LT) and how to implement LT processing in your application.
- [Filter and Expression Languages](#) on page 99, discusses the use of filter expressions to specify to an FDO provider how to identify a subset of the objects of an FDO data store.
- [The Geometry API](#) on page 111, discusses the various Geometry types and formats and describes how to work with the Geometry API to develop FDO-based applications.
- [Autodesk FDO Provider for Oracle](#) on page 123, discusses development issues that apply when using FDO Provider for Oracle.
- [OSGeo FDO Provider for ArcSDE](#) on page 147, discusses development issues that apply when using FDO Provider for ESRI® ArcSDE®.
- [OSGeo FDO Provider for MySQL](#) on page 165, discusses development issues that apply when using FDO Provider for MySQL.
- [OSGeo FDO Provider for ODBC](#) on page 173, discusses development issues that apply when using FDO Provider for ODBC.
- [Autodesk FDO Provider for Raster](#) on page 179, discusses development issues that apply when using FDO Provider for Raster.
- [OSGeo FDO Provider for SDF](#) on page 187, discusses development issues that apply when using FDO Provider for SDF.
- [OSGeo FDO Provider for SHP](#) on page 195, discusses development issues that apply when using FDO Provider for SHP (Shape).
- [Autodesk FDO Provider for SQL Server](#) on page 201, discusses development issues that apply when using FDO Provider for SQL Server.

- [OSGeo FDO Provider for WFS](#) on page 209, discusses development issues that apply when using FDO Provider for WFS.
- [OSGeo FDO Provider for WMS](#) on page 215, discusses development issues that apply when using FDO Provider for WMS.
- [Expression Functions](#) on page 221, outlines the signatures and implementation details for the enhanced expression functions.

What's New

This section summarizes the changes and enhancements you will find in this version of FDO.

Enhanced Set of Expression Functions

The enhanced set includes aggregate, conversion, date, mathematical, numeric, string and geometry functions. All functions are supported by all providers, with the exception of the Raster, WFS and WMS providers.

For more information and implementation details about the expression function signatures, the RDBMS-specific built-in support for some of the functions, and the provider-specific support, see the appendix [Expression Functions](#) on page 221.

Introduction

2

You can use the APIs in the FDO API to manipulate, define, and analyze geospatial information. This chapter introduces application development with the FDO API and explains the role of a feature provider.

What Is the FDO API?

From the Perspective of the Client Application User

The FDO API is a set of APIs used for creating, managing, and examining information, enabling Autodesk GIS products to seamlessly share spatial and non-spatial information, with minimal effort.

FDO is intended to provide consistent access to feature data, whether it comes from a CAD-based data source, or from a relational data store that supports rich classification. To achieve this, FDO supports a model that can readily support the capabilities of each data source, allowing consumer applications functionality to be tailored to match that of the data source. For example, some data sources may support spatial queries, while others do not. Also, a flexible metadata model is required in FDO, allowing clients to adapt to the underlying feature schema exposed by each data source.

From the Perspective of the Client Application Engineer

The FDO API provides a common, general purpose abstraction layer for accessing geospatial data from a variety of data sources. The API is, in part, an interface

specification of the abstraction layer. A provider, such as Autodesk FDO Provider for Oracle, is an implementation of the interface for a specific type of data source (for example, for an Oracle relational database). The API supports the standard data store manipulation operations, such as querying, updating, versioning, locking, and others. It also supports analysis.

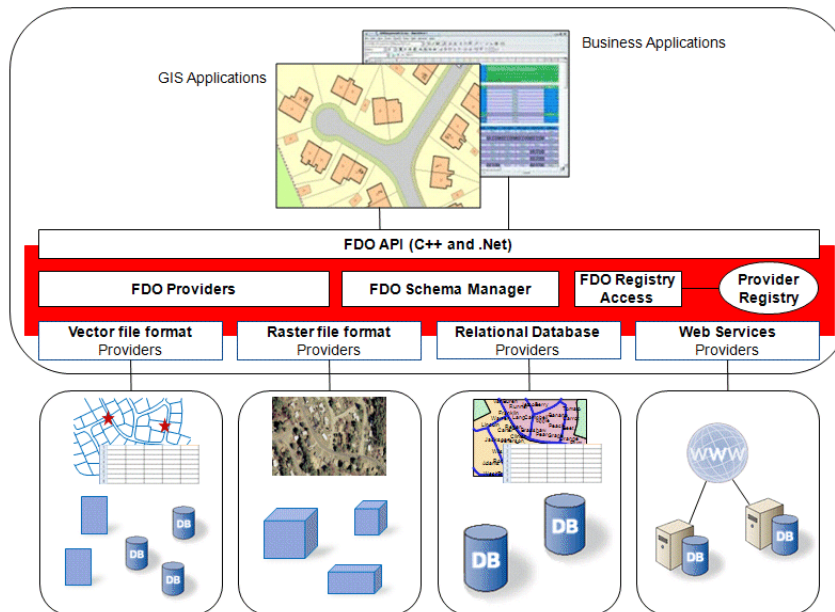
The API includes an extensive set of methods that return information about the capabilities of the underlying data source. For example, one method indicates whether the data source supports the creation of multiple schemas, and another indicates whether the data source supports schema modification.

A core set of services for providers is also available in the API, such as provider registration, schema management, filter and expression construction, and XML serialization and deserialization.

The API uses an object-oriented model for the construction of feature schema. A feature is a class, and its attributes, including its geometry, are a property of the class. The instantiation of a feature class, a Feature Data Object (FDO), can contain other FDOs.

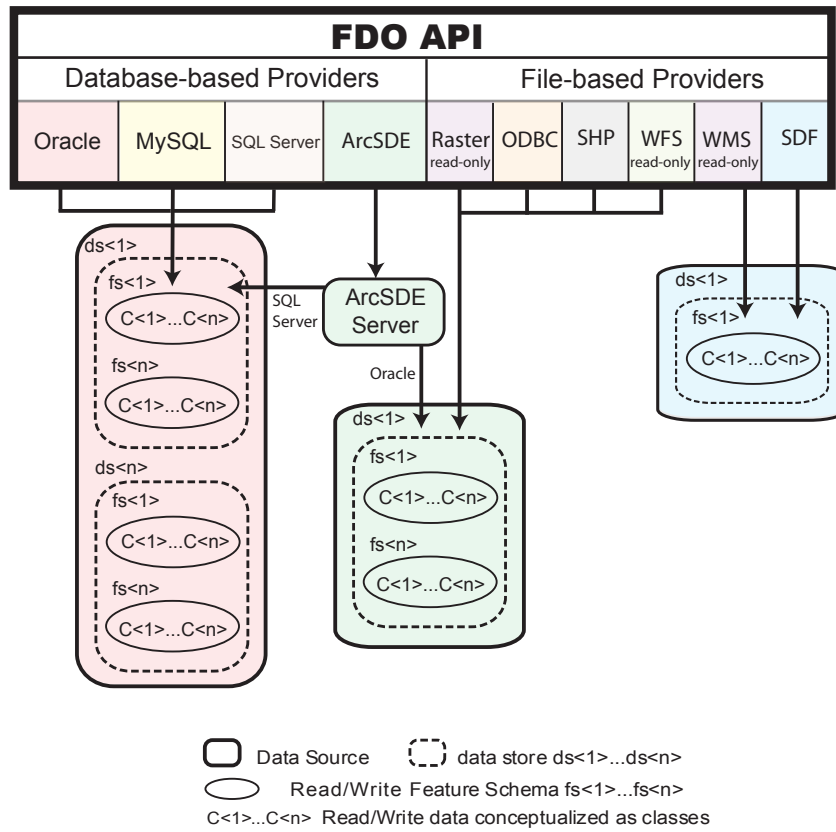
Getting Started

For detailed information to help you install and get started using Feature Data Objects (FDO), see *The Essential FDO*. It provides details about connecting to and configuring providers, data store management (create/delete), user IDs (create, grant permissions), and spatial context.



FDO Architecture and Providers

The following diagram shows the high-level overview architecture of the FDO API and included FDO providers.



FDO Architecture and Providers

FDO Packages

FDO is assembled in conceptual packages of similar functionality. This conceptual packaging is reflected in the substructure of the FDO SDK “includes” folder. For more information about the structure, see *The Essential FDO*.

FDO commands, provider-specific commands, and connections and capabilities provide access to native data stores through each different FDO provider. Schema management (through XML), client services, and filters and expressions are provider-independent packages that tie into the FDO API. Each of these are explained in more detail in subsequent sections.

The FDO API consists of classes grouped within the following packages:

- **Commands package.** Contains a collection of classes that provide the commands allowing the application to select and update features, define new types of feature classes, lock features, and perform analysis on features. Each Command object executes a specific type of command against the underlying data store. In addition, FDO providers expose one or more Command objects.
- **Connections/Capabilities.** Contains a collection of classes that establish and manage the connection to the underlying data store. Connection objects implement the `FdoIConnection` interface. Capabilities API provides the code for retrieving the various FDO provider capability categories, such as connection or schema capabilities. You can use this this API to determine the capabilities of a particular provider.
- **Filters and Expression package.** Contains a collection of classes that define filters and expression in FDO, which are used to identify a subset of objects of an FDO data store.
- **Client Services package.** Contains a collection of classes that define the client services in FDO that, for example, enable support for dynamic creation of connection objects given a provider name.
- **Schema package and FDO XML.** Contains a collection of classes that provides a logical mechanism for specifying how to represent geospatial features. The FDO feature schema is based somewhat on a subset of the OpenGIS and ISO feature models. FDO feature schemas can be written to an XML file. The `FdoFeatureSchema` and `FdoFeatureSchemaCollection` classes support the `FdoXmlSerializable` interface.

In addition, FDO is integrated with the Geometry API, which includes the classes that support specific Autodesk applications and APIs, including FDO.

For more information about each of the FDO packages, see *FDO API Reference Help* (FDO_API.chm) and subsequent chapters in this guide.

Provider API(s) complete the FDO API configuration. Each provider has a separate API reference Help (for example, `SDF_Provider_API.chm`).

What Is a Provider?

A provider is a specific implementation of the FDO API. It is the software component that provides access to data in a particular data store.

For this release, the providers that are included are as follows:

NOTE Autodesk FDO Provider for Oracle and FDO Provider for ArcSDE are listed first because they were included in previous releases. The remaining providers are new to this release and are in alphabetical order. Providers referenced in this document with “Autodesk” as part of their name are included only with Autodesk software. Other providers are open source. For more information, see the Open Source Geospatial Foundation website at www.OSGeo.org.

- **Autodesk FDO Provider for Oracle.** Read/write access to feature data in an Oracle-based data store. Supports spatial indexing, long transactions, and persistent locking. A custom API can gather provider information, transmit client services exceptions, list data stores, and create connection objects.
- **OsGeo FDO Provider for ArcSDE.** Read/write access to feature data in an ESRI ArcSDE-based data store (that is, with an underlying Oracle or SQL Server database). Supports describing schema, and inserting, selecting, updating, and deleting feature data in existing schemas; does not support creating or deleting schemas.
- **OsGeo FDO Provider for MySQL.** Read/write access to feature data in a MySQL-based data store. Supports spatial data types and spatial query operations. A custom API can gather information, transmit exceptions, list data stores, and create connection objects. MySQL architecture supports various storage engines, characteristics, and capabilities.
- **OsGeo FDO Provider for ODBC.** Read/write access to feature data in an ODBC-based data store. Supports XYZ feature objects and can define feature classes for any relational database table with X, Y, and optionally Z columns; does not support creating or deleting schema. Object locations are stored in separate properties in the object definition.
- **Autodesk FDO Provider for Raster.** Read-only access to feature data in raster-based file format. Supports various image and GIS data formats (for example, JPEG, PNG, MrSID, and others). Supports georeferenced file-based raster images and file-based grid coverages. Pixel-based images, such as satellite images, are useful underneath vector data.
- **OsGeo FDO Provider for SDF.** Read-write access to feature data in an SDF-based data store. Autodesk’s geospatial file format, SDF, supports multiple features/attributes, provides high performance for large data sets and interoperability with other Autodesk products, and spatial indexing. The SDF provider a valid alternative to database storage. Note that this release of the SDF provider supports version 3.0 of the SDF file format.

- **OsGeo FDO Provider for SHP.** Read/write access to existing spatial and attribute data in an ESRI SHP-based data store, which consists of separate shape files for geometry, index, and attributes. Each SHP and its associated DBF file is treated as a feature class with a single geometry property. This is a valid alternative to database storage but does not support locking.
- **Autodesk FDO Provider for SQL Server.** Read/write access to feature data in a Microsoft SQL Server-based data store. A custom API supports schema read/write access, and geospatial and non-geospatial data read/write access.
- **OsGeo FDO Provider for WFS.** Read-only access to feature data in an OGC WFS-based data store. Supports client/server environment and retrieves geospatial data encoded in GML from one or more Web Feature Services sites. Client/server communication is encoded in XML with the exception of feature geometries, which are encoded in GML. Note that there is no public API documentation for this provider; all functionality is accessible via the base FDO API.
- **OsGeo FDO Provider for WMS.** Read-only access to feature data in an OGC WMS-based data store. Web Map Service (WMS) produces maps of spatially referenced data dynamically from geographic information, which are generally rendered in PNG, GIF, or JPEG, or as vector-based Scalable Vector Graphics (SVG) or Web Computer Graphics Metafile (WebCGM) formats.

FDO supports retrieval and update of spatial and non-spatial GIS feature data through a rich classification model that is based on OpenGIS and ISO standards.

An overview of the relationships between providers, data sources, data stores, and schemas is presented in the [FDO Architecture and Providers](#) on page 7 graphic.

For more detailed information about the providers, see the appropriate appendix in this document. Data sources and data stores are discussed in the [Establishing a Connection](#) on page 27 chapter. Schema concepts are discussed in the [Schema Management](#) on page 47 chapter.

Developing Applications

You will need to perform several major tasks in using the FDO API to develop a custom application. Each of these tasks breaks down into a number of more detailed coding issues.

The major development tasks are:

- Working with the Build Environment
- Establishing a Connection
- Schema Management
- Data Maintenance
- Creating Queries
- Using Custom Commands (Provider-Specific)

These tasks are explored in detail in the chapters that follow.

FDO Concepts

3

Before you can work properly with the FDO API, you need to have a good understanding of its basic, underlying concepts. This chapter defines the essential constructs and dynamics that comprise the FDO API. The definitions of these constructs and dynamics are grouped into two interdependent categories:

- **Data Concepts.** Definitions of the data constructs that comprise the FDO API
- **Operational Concepts.** Definitions of the operations that are used to manage and manipulate the data.

Data Concepts

All concepts that are defined in this section relate to the data that FDO is designed to manage and manipulate.

What Is a Feature?

A feature is an abstraction of a natural or man-made real world object. It is related directly or indirectly to geographic locations. A spatial feature has one or more geometric properties. For example, a road feature might be represented by a line, and a hydrant might be represented by a point. A non-spatial feature does not have geometry, but can be related to a spatial feature which does. For example, a road feature may contain a sidewalk feature that is defined as not containing a geometry.

What Is a Schema?

A schema is a logical description of the data types used to model real-world objects. A schema is not the actual data instances (that is, not a particular road or land parcel), rather it is metadata. A schema is a model of the types of data

that would be found in a data store. For example, a schema which models the layout of city streets has a class called Road, and this class has a property called Name. The definition of Road and its associated classes constitute the schema.

What Is a Schema Override?

A schema override comprises instructions to override the default schema mappings. For example, an RDBMS-type FDO provider could map a feature class to a table of the same name by default. A schema override might map the class to a differently named table, for example, by mapping the "pole" class to the "telco_pole" table.

What is a Schema Mapping

A Schema Mapping is a correspondence between a Schema Element and a physical object in a data store. For example, OSGeo FDO Provider for MySQL maps each Feature Class onto a table in the MySQL database where the data store resides. The physical structure of data stores for each FDO provider can vary greatly, so the types of Schema Mappings can also vary between providers. Each provider defines a set of default schema mappings. For example, OSGeo FDO Provider for MySQL maps a class to a table of the same name by default. These defaults can be overridden by specifying Schema Overrides.

What Are Elements of a Schema?

A schema consists of a collection of schema elements. In the FDO API, schema elements are related to one another by derivation and by aggregation. An element of a schema defines a particular type of data, such as a feature class or a property, or an association. For example, a feature class definition for a road includes the class name (for example, Road), and the class properties (for example, Name, NumberLanes, PavementType, and Geometry).

What Is a Class Type?

A class type is a specialization of the base FDO class definition (FdoClassDefinition). It is used to represent the complex properties of spatial and non-spatial features.

What is a Feature Class?

A feature class is a schema element that describes a type of real-world object. It includes a class name and property definitions, including zero or more geometric properties. It describes the type of data that would be included in object instances of that type.

What Is a Property?

A property is a single attribute of a class and a class is defined by one or more property definitions. For example, a Road feature class may have properties called Name, NumberLanes, or Location. A property has a particular type, which can be a simple type, such as a string or number, or a complex type defined by a class, such as an Address type, which itself is defined by a set of properties, such as StreetNumber, StreetName, or StreetType.

There are five kinds of properties: association properties, data properties, geometric properties, object properties, and raster properties.

Individual properties are defined in the following sections.

What Is an Association Property?

The FdoAssociationPropertyDefinition class is used to model a peer-to-peer relationship between two classes. This relationship is defined at schema creation time and instantiated at object creation time. The association property supports various cardinality settings, cascading locks, and differing delete rules. An FDO filter can be based on association properties and FdoIFeatureReader can handle associated objects through the GetObject() method.

What Is a Data Property?

A data property is a non-spatial property. An instance of a data property contains a value whose type is either boolean, byte, date/time, decimal, single, double, Int16, Int32, Int64, string, binary large object, or character large object.

What Is Dimensionality?

Dimensionality, and the concept of dimension, has two different meanings in the discussion of geometry and geometric property.

The first is called shape dimensionality, and it is defined by the FdoGeometricType enumeration. The four shapes are point (0 dimensions), curve (1 dimensions), surface (2 dimensions), and solid (3 dimensions).

The other is called ordinate dimensionality, and it is defined by the FdoDimensionality enumeration. There are four ordinate dimensions: XY, XYZ, XYM, and XYZM. M stands for measure.

What Is a Geometric Property?

An instance of a geometric property contains an object that represents a geometry value. The definition of the geometric property may restrict an object to represent a geometry that always has the same shape, such as a point, or it could allow different object instances to have different dimensions. For example, one geometric property object could represent a point and another could represent a line. Any combination of shapes is permissible in the specification of the geometric types that a geometry property definition permits. The default geometric property specifies that an object could represent a geometry that is any one of the four shapes.

With respect to ordinate dimensionality, all instances of a geometric property must have the same ordinate dimension. The default is XY.

Geometric property definitions have two attributes regarding ordinate dimensionality: HasElevation for Z and HasMeasure for M.

What is a Geometry?

A geometry is represented using geometric constructs either defined as lists of one or more XY or XYZ points or defined parametrically, for example, as a circular arc. While geometry typically is two- or three-dimensional, it may also contain the measurement dimension (M) to provide the basis for dynamic segments.

The geometry types are denoted by the `FdoGeometryType` enumeration and describe the following:

- Point
- LineString (one or more connected line segments, defined by positions at the vertices)
- CurveString (a collection of connected circular arc segments and linear segments)
- Polygon (a surface bound by one outer ring and zero or more interior rings; the rings are closed, connected line segments, defined by positions at the vertices)
- CurvePolygon (a surface bound by one outer ring and zero or more interior rings; the rings are closed, connected curve segments)
- MultiPoint (multiple points, which may be disjoint)
- MultiLineString (multiple LineStrings, which may be disjoint)

- MultiCurveString (multiple CurveStrings, which may be disjoint)
- MultiPolygon (multiple Polygons, which may be disjoint)
- MultiCurvePolygon (multiple CurvePolygons, which may be disjoint)
- MultiGeometry (a heterogeneous collection of geometries, which may be disjoint)

Most geometry types are defined using either curve segments or a series of connected line segments. Curve segments are used where non-linear curves may appear. The following curve segment types are supported:

- CircularArcSegment (circular arc defined by three positions on the arc)
- LineStringSegment (a series of connected line segments, defined by positions are the vertices)

There are currently no geometries of type “solid” (3D shape dimensionality) supported.

The `FdoIConnection::GetGeometryCapabilities()` method can be used to query which geometry types and ordinate dimensionalities are supported by a particular provider.

What Is an Object Property?

An object property is a complex property type that can be used within a class, and an object property, itself, is defined by a class definition. For example, the Address type example described previously in the Property definition. An object property may define a single instance for each class object instance (for example, an address property of a land parcel), or may represent a list of instances of that class type per instance of the owning class (for example, inspection records as a complex property of an electrical device feature class).

What is a Raster Property?

A raster property defines the information needed to process a raster image, for example, the number of bits of information per pixel, the size in pixels of the X dimension, and the size in pixels of the Y dimension, needed to process a raster image.

What Is a Spatial Context?

A spatial context describes the general metadata or parameters within which geometry for a collection of features resides. In particular, the spatial context includes the definition of the coordinate system, spheroid parameters, units, spatial extents, and so on for a collection of geometries owned by features.

Spatial context can be described as the “coordinate system plus identity.” Any geometries that are to be spatially related must be in a common spatial context.

The identity component is required in order to support separate workspaces, such as schematic diagrams, which are non-georeferenced. Also, it supports georeferenced cases. For example, two users might create drawings using some default spatial parameters (for example, rectangular and 10,000x10,000), although each drawing had nothing to do with the other. If the drawings were put into a common database, the users could preserve not only the spatial parameters, but also the container aspect of their data, using spatial context.

For more information about spatial context, see [Spatial Context](#) on page 120.

What is a Data Store?

A data store is a repository of an integrated set of objects. The objects in a data store are modeled either by classes or feature classes defined within one or more schemas. For example, a data store may contain data for both a LandUse schema and a TelcoOutsidePlant schema. Some data stores can represent data in only one schema, while other data stores can represent data in many schemas (for example, RDBMS-based data stores, such as MySQL).

Operational Concepts

The concepts that are defined in this section relate to the FDO operations used to manage and manipulate data.

What Is a Command?

In FDO, the application uses a command to select and update features, define new types of feature classes, lock features, version features, and perform some analysis of features. Each Command object executes a specific type of command against the underlying data store. Interfaces define the semantics of each command, allowing them to be well-defined and strongly typed. Because FDO uses a standard set of commands, providers can extend existing commands and add new commands, specific to that provider. Feature commands execute

against a particular connection and may execute within the scope of a transaction.

An FDO command is a particular FDO interface that is used by the application to invoke an operation against a data store. A command may retrieve data from a data store (for example, a Select command), may update data in a data store (for example, an Update or Delete command), may perform some analysis (for example, an Activate Spatial Context command), or may cause some other change in a data store or session (for example, a Begin Transaction command).

What Is an Expression?

An expression is a construct that an application can use to build up a filter. An expression is a clause of a filter or larger expression. For example, “Lanes >=4 and PavementType = 'Asphalt'” takes two expressions and combines them to create a filter.

For more information about using expressions with FDO, see [Filter and Expression Languages](#) on page 99.

What Is a Filter?

A filter is a construct that an application specifies to an FDO provider to identify a subset of objects of an FDO data store. For example, a filter may be used to identify all Road type features that have 2 lanes and that are within 200 metres of a particular location. Many FDO commands use filter parameters to specify the objects to which the command applies. For example, a Select command uses a filter to identify the objects that the application wants to retrieve. Similarly, a Delete command uses a filter to identify the objects that the application wants to delete from the data store.

For more information about using filters with FDO, see [Filter and Expression Languages](#) on page 99.

What Is Locking?

A user can use locking to gain update control of an object in the data store to the exclusion of other users. There are two general types of locks—transaction locks and persistent locks. Transaction locks are temporary and endure only for the duration of the transaction (see [What Is a Transaction?](#) on page 20).

Persistent locks applied to objects by a user remain with the object until either that user removes those locks or the locks are removed by another user with the appropriate authority.

What Is a Transaction?

A transaction changes the data store in some way. The way these changes affect the data store is determined by the transaction's properties. For example, the Atomic property specifies that either all changes happen or non happen. In transaction processing the data store treats a series of commands as a single atomic unit of change to that data store. Either all changes generated by the commands are successful or the whole set is cancelled. A transaction is a single atomic unit of changes to a data store. The application terminates a transaction with either a "commit," which applies the set of changes, or a "rollback," which cancels the set of changes. Further, the data store may automatically roll back a transaction if it detects a severe error in any of the commands within the transaction. A transaction has the following properties:

- **Atomic.** Either all changes generated by the commands within a transaction happen or none happen.
- **Consistent.** The transaction leaves the data store in a consistent state regarding any constraints or other data integrity rules.
- **Isolated.** Changes being made within a transaction by one user are not visible to other users until after that transaction is committed.
- **Durable.** After a transaction is completed successfully, the changes are persistent in the data store on disk and cannot be lost if the program or processor fails.

What Is a Long Transaction?

A long transaction (LT) is an administration unit used to group conditional changes to objects. Depending on the situation, such a unit might contain conditional changes to one or to many objects. Long transactions are used to modify as-built data in the database without permanently changing the as-built data. Long transactions can be used to apply revisions or alternates to an object.

What Is a Root Long Transaction?

A root long transaction is a long transaction that represents permanent data. Any long transaction has a root long transaction as an ancestor in its long transaction dependency graph.

Development Practices

4

This chapter explains several practices to follow when working with the FDO API and provides examples of how to follow these practices.

Memory Management

Some FDO functions (for example, the Create methods) allocate memory when they are called. This memory needs to be freed to prevent memory leaks. All destructors on FDO classes are protected, so you must call a Release() function to destroy them (thus freeing their allocated memory). Each class inherits from the `FdoIDisposable` class, which defines the Release() method and the AddRef() method.

In addition, these classes are reference counted, and the count is increased (by AddRef()) when you retrieve them through a Get function. After finishing with the object, you need to release it (just as with COM objects). The object is destroyed only when the reference count hits 0. Two macros are defined to help in the use of the Release() and AddRef() methods.

FDO_SAFE_RELEASE (*ptr)

If the “*ptr” argument is not null, FDO_SAFE_RELEASE calls the release() method of the object pointed to by the “*ptr” argument and then sets the local pointer to the object to NULL. The macro definition is `#define FDO_SAFE_RELEASE(x)`

```
{if (x) (x)->Release(); (x) = NULL;};
```

```
FdoFeatureClass* pBase = myClass->GetBaseClass();  
...  
// Must release reference added by GetBaseClass when done.  
FDO_SAFE_RELEASE(pBase);
```

FDO_SAFE_ADDREF (*ptr)

If the “*ptr” argument is not null, FDO_SAFE_ADDREF calls the AddRef() method of the object pointed to by the “*ptr” argument. The macro definition is `#define FDO_SAFE_ADDREF(x) ((x != NULL) ? (x)->AddRef(), (x):(NULL))`.

- `return FDO_SAFE_ADDREF(value)` returns NULL if value equals NULL or increments the reference count of the object that value points to and returns value.
- `m_list[index] = FDO_SAFE_ADDREF(value)` assigns NULL to the array entry if value is NULL or increments the reference count of the object that value points to and assigns value to the array entry.

FdoPtr

An `FdoPtr` smart pointer is provided to help manage memory. You wrap an FDO object in a `FdoPtr`. The requirement is that the object’s type must inherit from `FdoIDisposable`. The object is then released automatically when the `FdoPtr` goes out of scope. The following code illustrates how to use `FdoPtr`:

```
FdoPtr<FdoFeatureClass> pBase = myClass->GetBaseClass();  
...  
// No need to call FDO_SAFE_RELEASE.  
// Before it is destroyed, pBase calls Release() on the FdoFeature  
Class object
```

NOTE If, for some reason, you wanted to use `FDO_SAFE_RELEASE` on an `FdoPtr`, you would have to use an `FdoPtr` method to get a pointer to the object that `FdoPtr` wraps and pass that pointer to `FDO_SAFE_RELEASE`.

You can use `FdoPtr` for your own classes by inheriting from the abstract class `FdoIDisposable` and providing an implementation for the `Dispose()` method (typically `delete this;`).

FdoPtr Typedefs

Typedefs are provided that define identifiers representing Fdo classes wrapped by `FdoPtr`. An example is `typedef FdoPtr<FdoClass> FdoClassP`.

Collections

You can use FDO collection template classes to store your own objects. The requirements for your collection class and the class used to instantiate the template are the same as those for wrapping a class in a `FdoPtr`.

Exception Handling

In the FDO API, `FdoCommandException` class is the exception type thrown from classes in the `Commands` package, and `FdoConnectionException` class is the exception type thrown from classes in the `Connections` package. Both of these exception types derive from a language-level exception class that is environment-specific.

All exceptions are derived from the `FdoException` class. To catch and process specific exception types, nest catch statements as in the following example:

```
try {
    ... code
}
catch (FdoCommandException *ex) {
    .. process message
}
catch (FdoException *ex) {
    .. process message
}
```

In some cases, underneath an FDO command, the FDO level throws an `FdoException`. The FDO command then traps the `FdoException` and wraps it in an `FdoCommandException` (or `FdoSchemaException` for a schema command). In this case, several messages are returned by one exception. The following example shows how to process multiple messages from one exception:

```
catch ( FdoSchemaException* ex ) {
    // Loop through all the schema messages
    FdoException* currE = ex;
    while ( currE ) {
        CW2A msg(currE->GetExceptionMessage());
        acutPrintf ("FdoConnectionException: %s\n", msg);
        currE = currE->GetCause();
    }
}
```

An application function may need to catch and then re-throw exceptions in order to clean up memory. However, the need to do this can be eliminated by using `FdoPtr`. The following example cleans up memory on error:

```
FdoFeatureClass* pBase = NULL;
try {
    pBase = myClass->GetBaseClass();
    ...
}
catch (...) {
    FDO_SAFE_RELEASE(pBase);
    throw;
}
// Must release reference added by GetBaseClass when done.
FDO_SAFE_RELEASE(pBase);
```

The catch and rethrow is unnecessary when `FdoPtr` is used:

```
FdoPtr<FdoFeatureClass> pBase = myClass->GetBaseClass();
...
```

Exception Messages

Exception messages are localized. On Windows the localized strings are in resource-only DLLs, and on Linux they are in catalogs. The message DLLs are in the *bin* folder; the DLL name contains Message or Msg. The catalog files are in the */usr/local/fdo-3.2.0/nls* directory; the names of these files ends in *.cat*. NLS stands for National Language Support.

On Linux set the `NLSPATH` environment variable so that the runtime code can locate the message catalogs. For example, `export NLSPATH=/usr/local/fdo-3.2.0/nls/%N`.

On Windows you do not have to do anything special to enable the runtime code to locate the message DLLs.

The contents of the exception message files are indexed. When you call one of the `FdoException::NLGetMessage` methods declared in `Exception.h`, you provide a message number argument. You may also provide a default message string argument. In the event that the exception message resource file cannot be found, the default message is substituted instead. If the default message string is not provided and the resource file cannot be found, the message number is used as the exception message. Not finding the resource file can only happen on Linux and only if the `NLSPATH` environment variable is not set.

The following two examples, when called on Linux with the NLSPATH environment variable not set, show the use of the default message and the message number in the exception message.

The following is an example of using the default string: throw

```
FdoSchemaException::Create(NlsMsgGet1(FDORDBMS_333, "Class '%1$s' not found", value->GetText()));
```

The following is an example of not setting the default string and using the message number instead: FdoSchemaException* pNewException =

```
FdoSchemaException::Create(  
FdoSmError::NLSGetMessage(FDO_NLSID(FDOSM_221),  
pFeatSchema->GetName()), pException);
```

Managing FdoPtr Behaviors

The topics in this section describe several ways that you can manager FdoPtr behavior. For more information about managing FdoPtr behavior, see the related topics “FdoPtr <T> Class Template Reference” and “FdoIDisposable Class Reference” in the *FDO Reference Help* and *The Essential FDO*.

Chain Calls

Do not chain calls. If you do, returned pointers will not be released. For example, given an FdoClassDefinition* pclassDef:

```
psz = pclassDef ->GetProperties()->GetItem(0)->GetName();
```

The above code would result in two memory leaks. Instead use:

```
FdoPropertyDefinitionCollection* pprops = pclassDef -> GetProperties();  
FdoPropertyDefinition* ppropDef = pprops->GetItem(0);  
psz = ppropDef->GetName();  
ppropDef->Release();  
pprops->Release();
```

or (with FdoPtr):

```
FdoPtr<FdoPropertyDefinitionCollection> pprops = pclassDef-> GetProperties();  
FdoPtr<FdoPropertyDefinition> ppropDef = pprops-> GetItem(0);  
psz = ppropDef->GetName();
```

or (also with FdoPtr):

```
psz = FdoPtr <FdoPropertyDefinition> (FdoPtr <FdoPropertyDefini
tionCollection>(pclassDef->GetProperties())-> GetItem(0))->Get
Name();
```

Assigning Return Pointer of an FDO Function Call to a Non-Smart Pointer

If you are assigning the return pointer of an FDO function call to a non-smart pointer, then you should assign that pointer to a FdoPtr. For example:

```
FdoLineString* P = gf.CreateLineString(...);
FdoPtr <FdoLineString> p2 = FDO_SAFE_ADDREF(p);
```

Establishing a Connection

5

This chapter explains how to establish a connection to an FDO provider and provides a connection example.

Connection Semantics

Data Sources and Data Stores

The FDO API uses connection semantics to implement access to feature schema data. The term data store is used to refer to a collection of zero or more objects, which instantiate class definitions belonging to one or more FDO feature schema. The connection is to a data store because that is where data objects are stored. The underlying data source technologies used to hold data stores can be relational databases, such as, a MySQL database, or a file-based solution, such as an SDF file.

The mapping of a data store to data source technology can be one-to-one or many-to-one. For example, it is

- One-to-one when the connection is made by way of the OSGeo FDO Provider for ArcSDE and the ArcSDE server is using an Oracle database.
- Many-to-one when the data source is a MySQL database and the connection is made by way of the OSGeo FDO Provider for MySQL (in this case, the data store is like a container within a container).

When many-to-one mapping is possible, a connection can be made in one or two steps. For more information, see [Establishing a Connection](#) on page 29 and *The Essential FDO*.

The underlying data source technologies differ in the connection parameters used for connecting to a particular provider. The values for these parameters are generated during the installation and configuration of the container technologies. For more information about these values and the process of installing and configuring the associated data source technologies, see the appropriate appendix in this document and *The Essential FDO*.

Providers

You connect to a data store by way of a provider.

The FDO API contains a registry interface that you can use to register or deregister a provider. See the class `FdoProviderRegistry` in *Inc/Fdo/ClientServices/ProviderRegistry.h*.

The providers are registered during the initialization of the FDO SDK. In order to connect to a provider, you will need the name of the provider in a particular format: `<Company/Foundation/Originator>.<Provider>.<Version>`. The `<Company/Foundation/Originator>` and `<Provider>` values are invariable. For specific values, see *The Essential FDO*.

In order to connect, you will need the full name including the `<Version>` value. You can retrieve the full name from the registry and display the set of provider names in a connection menu list. If, for whatever reason, you deregister a provider, save the registry information for that provider in case you want to reregister it again. The provider object returned by the registry has a `Set()` method to allow you to change values. However, the only value you can safely change is the display name. Sample code for using the provider registry is located in [Establishing a Connection](#) on page 29.

The registry contains the following information about a provider:

- **Name.** The unique name of the feature provider. This name should be of the form `<Company/Foundation/Originator>.<Provider>.<Version>`, for example, `Autodesk.Oracle.3.0` or `OSGeo.MySQL.3.0`.
- **DisplayName.** A user-friendly display name of the feature provider. The initial values of this property for the pre-registered providers are “Autodesk FDO Provider for Oracle”, “OSGeo FDO Provider for SDF”, etc., or the equivalent in the language of the country where the application is being used.
- **Description.** A brief description of the feature provider. For example, the the OSGeo FDO Provider for SDF description is “Read/write access to Autodesk's spatial database format, a file-based personal geodatabase that supports multiple features/attributes, spatial indexing, and file-locking.”

- **Version.** The version of the feature provider. The version number string has the form <VersionMajor>.<VersionMinor>.<BuildMajor>.<BuildMinor>, for example, 3.0.0.0.
- **FDOVersion.** The version of the feature data objects specification to which the feature provider conforms. The version number string has the form <VersionMajor>.<VersionMinor>.<BuildMajor>.<BuildMinor>, for example, 3.0.1.0.
- **libraryPath.** The FULL library path including the library name of the provider, for example, <FDO SDK Install Location>/bin/FdoRdbms.dll.
- **isManaged.** A flag indicating whether the provider is a managed or unmanaged .NET provider.

Establishing a Connection

As mentioned in a previous section, [Connection Semantics](#) on page 27, the FDO API uses a provider to connect to a data store and its underlying data source technology. These data source technologies must be installed and configured. Certain values generated during data source installation and configuration are used as arguments during the connection process. Because the FDO API does not provide any methods to automate the collection and presentation of these configuration values, either the application developer must request the user to input these configuration values during the connection process, or the application developer can provide an application configuration interface, which would populate the application with the container configuration values and thus allow the user to choose them from lists.

NOTE For more information about connecting, see *The Essential FDO*.

A connection can be made in either one or two steps:

- **One-step connection.** If the user sets the required connection properties and calls the connection object's `Open()` method once, the returned state is `FdoConnectionState_Open`, no additional information is needed.
- **Two-step connection.** If the user sets the required connection properties and calls the connection object's `Open()` method, the returned state is `FdoConnectionState_Pending`, additional information is needed to complete the connection. In this case, the first call to `Open()` has resulted in the retrieval of a list of values for a property that becomes a required property

for the second call to the `Open()` method. After the user has selected one of the values in the list, the second call to `Open()` should result in `FdoConnectionState_Open`.

Connecting to a data store by way of the MySQL or the ArcSDE provider, for example, can be done in either one or two steps. In the first step, the data store parameter is not required. If the user does not give the data store parameter a value, the FDO will retrieve the list of data store values from the data source so that the user can choose from them during the second step. Otherwise the user can give the data store a value in the first step, and assuming that the value is valid, the connection will be completed in one step.

For the purpose of this example, let's assume that the user has installed MySQL on his local machine. During the installation he was prompted to assign a password to the system administrator account whose name is 'root'. He set the password to 'test'.

The following steps are preliminary to establishing a connection:

- 1 Get the list of providers.

```
FdoPtr<FdoProviderRegistry> registry = (FdoProviderRegistry
*)FdoFeatureAccessManager::GetProviderRegistry();
FdoProviderCollection * providers = registry->GetProviders();
```

- 2 Get the display names for all of the providers in the registry. An example of a display name might be OSGeo FDO Provider for MySQL.

```
FdoStringP displayName;
FdoStringP internalName;
FdoPtr<FdoProvider> provider;
int count = providers->GetCount();
for(int i = 0; i < count; i++) {
    provider = providers->GetItem(i);
    internalName = provider->GetName();
    displayName = provider->GetDisplayName();
    // add displayName to your list
}
```

- 3 Use the display names to create a menu list, from which the user will select from when making a connection.

After the user initiates a provider display name from the connection menu, do the following:

- 1 Loop through the providers in the registry until you match the display name selected by the user from the connection menu with a provider display name in the registry and retrieve the internal name for that provider. An example of an internal could be OSGeo.MySQL.3.2.

```
FdoStringP internalName = provider->GetName();
```

- 2 Get an instance of the connection manager.

```
FdoPtr<FdoConnectionManager> connectMgr = (FdoConnectionManager  
*) FdoFeatureAccessManager::GetConnectionManager();
```

- 3 Call the manager's CreateConnection() method using the provider internal name as an argument to obtain a connection object.

```
FdoPtr<FdoIConnection> fdoConnection = connectMgr->CreateConne  
ction(L"OsGeo.MySQL.3.2");
```

- 4 Obtain a connection info object by calling the connection object's GetConnectionInfo() method.

```
FdoPtr<FdoIConnectionInfo> info = fdoConnection->GetConnection  
Info();
```

- 5 Obtain a connection property dictionary object by calling the connection info object's GetConnection Properties() method and use this dictionary to construct a dialog box requesting connection information from the user.

```
FdoPtr<FdoIConnectionPropertyDictionary> ConnDict = info->GetCon  
nectionProperties();
```

- 6 Get a list of connection property names from the dictionary and use this list to get information about the property. The following code loops through the dictionary getting all of the possible information.

NOTE An attempt to get the values of an enumerable property is made only if the property is required.

```
FdoInt32 count = 0;  
FdoString ** names = NULL;  
FdoStringP name;  
FdoStringP localname;  
FdoStringP val;  
FdoStringP defaultVal;  
bool isRequired = false;
```

```

bool isProtected = false;
bool isFilename = false;
bool isFilepath = false;
bool isDatastorename = false;
bool isEnumerable = false;
FdoInt32 enumCount = 0;
FdoString ** enumNames = NULL;
FdoStringP enumName;
names = ConnDict->GetPropertyNames(count);
for(int i = 0; i < count; i++) {
    name = names[i];
    val = dict->GetProperty(name);
    defaultVal = dict->GetPropertyDefault(name);
    localname = dict->GetLocalizedName(name);
    isRequired = dict->IsPropertyRequired(name);
    isProtected = dict->IsPropertyProtected(name);
    isFilename = dict->IsPropertyFileName(name);
    isFilepath = dict->IsPropertyFilePath(name);
    isDatastorename = dict->IsPropertyDatastoreName(name);
    isEnumerable = dict->IsPropertyEnumerable(name);
    if (isEnumerable) {
        if (isRequired) {
            enumNames = dict->EnumeratePropertyValues(name, enumCount);
            for(int j = 0; j < enumCount; j++) {
                enumName = enumNames[j];
            }
        }
    }
}
}

```

- 7 Use the `GetLocalizedName` method to obtain the name of the property to present to the user. Calls to dictionary methods need the value of the internal name in the string array returned by `GetPropertyNames()`. So when the user selects the localized name in the menu, the program must map the localized name to the internal name.
- 8 Use the `IsPropertyRequired` method to determine whether to mark the line as either required or optional; the dialog box handler should not permit the user to click OK in the dialog box unless a required field has a value.
- 9 Use the `IsPropertyProtected` method to determine whether the dialog box handler should process the field value as protected data, for example, a password.

- 10 Use the `IsPropertyEnumerable` and `IsRequired` methods to determine whether to call the `EnumeratePropertyValues` method to get a list of valid values.

NOTE Call the `EnumeratePropertyValues` method only if both methods return true. Otherwise be prepared to catch an exception if there is no pending connection. The assumption is that a connection exists and the program is retrieving values from the data store.

As shown in the code lines above, the `EnumeratePropertyValues` method takes a property name and an updates integer argument and returns a string array. The updates integer will say how many values are in the returned array. Present the list of choices to the user.

If the property is not enumerable, present the values returned by either the `GetProperty` or `GetPropertyDefault` methods to the user.

Now that the user has seen the set of properties in the dictionary, s/he can set the required properties. A property is set by calling the dictionary's `SetProperty` method. The MySQL connection property names are `Username`, `Password`, `Service`, and `DataStore`. The dictionary tells us that `Username`, `Password`, and `Service` are required properties and that `DataStore` is not required. Let's connect to the MySQL as root.

```
ConnDict->SetProperty(L"Username", L"root");  
ConnDict->SetProperty(L>Password", L"test");  
ConnDict->SetProperty(L"Service", L"localhost");
```

NOTE `fdoconnection->GetConnectionString()` returns

```
Username=root;Password=test;Service=localhost;.
```

```
fdoconnection->SetConnectionString(L"Username=root;Password=test;Service=localhost;");
```

would set the connection properties to the same values as the three calls above to the connection dictionary's `SetProperty()` method.

Open the connection.

```
FdoConnectionState state = fdoConnection->Open();
```

The value of `state` is `FdoConnectionState_Pending`. An examination of the connection dictionary will reveal that the `DataStore` property is now required.

When the user checks the command capabilities, he discovers that he can create a data store.

```

FdoPtr<FdoICommandCapabilities> commandCapabilities = fdoConnection->GetCommandCapabilities();
bool bSupportsCreateDataStore = false;
FdoInt32 numCommands;
FdoInt32 * commands = commandCapabilities->GetCommands(numCommands);
for(int i = 0; i < numCommands; i++) {
    switch(commands[i]) {
        case FdoCommandType_CreateDataStore : bSupportsCreateDataStore = true;
    }
}

```

He can use the pending connection to MySQL to create the datastore. Use the connection object to create the `FdoICreateDataStore` command object. Use the command object to create the `FdoIDataStorePropertyDictionary` object and find out from this object what properties you must define. Use the dictionary object to set the required properties and then execute the command to create the 'fdo_user' data store. The only required property is `DataStore`.

NOTE The `FdoIDataPropertyDictionary` and the `FdoIConnectionPropertyDictionary` classes are both derived from `FdoIPropertyDictionary`. The code used above to access the `FdoIConnectionPropertyDictionary` object works for the `FdoIDataPropertyDictionary`.

```

FdoPtr<FdoICreateDataStore> createDataStoreCmd = dynamic_cast<FdoICreateDataStore *> (fdoConnection->CreateCommand(FdoCommandType_CreateDataStore));
FdoPtr<FdoIDataStorePropertyDictionary> createDsDict = createDataStoreCmd->GetDataStoreProperties();
createDsDict->SetProperty(L"DataStore", L"fdo_user");
createDataStoreCmd->Execute();

```

Now use the connection property dictionary to set the `DataStore` property to 'fdo_user' and call the `Open()` method on the connection object. This method should return `FdoConnectionState_Open`.

Capabilities

6

This chapter explains the Capabilities API and provides the code for retrieving the various FDO provider capability categories, such as connection or schema capabilities. You can use this this API to determine the capabilities of a particular provider.

What Is the Capabilities API?

You can use this API and its various capability categories to determine the capabilities of a particular provider, for example, FDO Provider for Oracle. The capabilities methods can be used to execute code conditionally, depending on which provider is being used and which capability is being exercised.

The capabilities of an FDO provider are grouped in the following categories:

- Connection
- Schema
- Command
- Expression
- Filter
- Geometry
- Raster

NOTE Topology-related samples are provided for informational use only. There is no interface or support provided. Autodesk reserves the right to change the software related to the content herein.

The capabilities are retrieved by using methods belonging to an `FdoIConnection` object. First, you connect to the provider. Then, you query its capabilities.

The sections in this chapter describe how to retrieve the capabilities for each of the categories. In each section, the code fragment assumes that you have connected to the provider and declared the following connection object:

```
#include <fdo.h>
FdoIConnection * connection;
```

Connection Capabilities

Code

Declarations

The object and variable declarations are the following:

```
FdoIConnectionCapabilities * connectionCapabilities;
// FdoThreadCapability is an enumerated type
FdoThreadCapability threadCapability;
// FdoSpatialContextExtentType is an enumerated type
FdoSpatialContextExtentType * spatialContextExtentTypes;
FdoInt32 numSpatialContexts;
// FdoLockType is an enumerated type
FdoLockType * lockTypes;
FdoInt32 numLockTypes;
bool supportsLocking;
bool supportsTimeout;
bool supportsTransactions;
bool supportsLongTransactions;
bool supportsSQL;
bool supportsConfiguration;
```

Method calls

The method calls are the following:

```

connectionCapabilities = connection->GetConnectionCapabilities();
// GetThreadCapability() returns a single value
threadCapability = connectionCapabilities->GetThreadCapability();
// GetSpatialContextTypes() returns a list
spatialContextExtentTypes = connectionCapabilities->
    GetSpatialContextTypes(numSpatialContexts);
// loop through the spatialContextExtentTypes
supportsLocking = connectionCapabilities->SupportsLocking();
// GetLockTypes() returns a list
lockTypes = connectionCapabilities->GetLockTypes(numLockTypes);
// loop through the lockTypes
supportsTimeout = connectionCapabilities->SupportsTimeout();
supportsTransactions = connectionCapabilities->
    SupportsTransactions();
supportsLongTransactions = connectionCapabilities->
    SupportsLongTransactions();
supportsSQL = connectionCapabilities->SupportsSQL();
supportsConfiguration = connectionCapabilities->
    SupportsConfiguration();

```

Reference

For more information, see these *FDO API Reference Help* topics:

- class FdoIConnectionCapabilities
- enum FdoLockType
- enum FdoSpatialContextExtentType
- enum FdoThreadCapability

Schema Capabilities

Code

Declarations

The object and variable declarations are the following:

```

FdoISchemaCapabilities * schemaCapabilities;
// FdoClassType is an enumerated type
FdoClassType * classTypes;
// FdoDataType is an enumerated type
FdoDataType * dataTypes;
bool supportsInheritance;
bool supportsMultipleSchemas;
bool supportsObjectProperties;
bool supportsAssociationProperties;
bool supportsSchemaOverrides;
bool supportsNetworkModel;
bool supportsAutoIdGeneration;
bool supportsDataStoreScopeUniqueIdGeneration;
FdoDataType * autoGeneratedTypes;
bool supportsSchemaModification;

```

Method Calls

The method calls are the following:

```

schemaCapabilities = connection->GetSchemaCapabilities();
// this returns a list of FdoClassType
classTypes = schemaCapabilities->GetClassTypes();
// loop through the classTypes
// this returns a list of FdoDataType
dataTypes = schemaCapabilities->GetDataTypes();
// loop through the dataTypes
supportsInheritance = schemaCapabilities->SupportsInheritance();
supportsMultipleSchemas = schemaCapabilities->
    SupportsMultipleSchemas();
supportsObjectProperties = schemaCapabilities->
    SupportsObjectProperties();
supportsAssociationProperties = schemaCapabilities->
    SupportsAssociationProperties();
supportsSchemaOverrides = schemaCapabilities->
    SupportsSchemaOverrides();
supportsNetworkModel = schemaCapabilities->SupportsNetworkModel();
supportsAutoIdGeneration = schemaCapabilities->
    SupportsAutoIdGeneration();
supportsDataStoreScopeUniqueIdGeneration = schemaCapabilities->
    SupportsDataStoreScopeUniqueIdGeneration();
// this returns a list of FdoDataType
autoGeneratedTypes = schemaCapabilities->
    GetSupportedAutoGeneratedTypes();
supportsSchemaModification = schemaCapabilities->
    SupportsSchemaModification();

```

References

For more information, see these *FDO API Reference Help* topics:

- class FdoISchemaCapabilities
- enum FdoClassType
- enum FdoDataType

Command Capabilities

Code

Declarations

The object and variable declarations are the following:

```
FdoICommandCapabilities * commandCapabilities;
// this will contain values of type FdoCommandType and
// possibly values of type FdoRdbmsCommandType, which are
// provider-specific commands
FdoInt32 * commandTypes;
bool supportsParameters;
bool supportsTimeout;
bool supportsSelectExpressions;
bool supportsSelectFunctions;
bool supportsSelectDistinct;
bool supportsSelectOrdering;
bool supportsSelectGrouping;
```

Method Calls

The method calls are the following:

```
commandCapabilities = connection->GetCommandCapabilities();
// this returns a list of command types
commandTypes = commandCapabilities->GetCommands();
// loop through the commandTypes
supportsParameters = commandCapabilities->SupportsParameters();
supportsTimeout = commandCapabilities->SupportsTimeout();
supportsSelectExpressions = commandCapabilities->
    SupportsSelectExpressions();
supportsSelectFunctions = commandCapabilities->
    SupportsSelectFunctions();
supportsSelectDistinct = commandCapabilities->
    SupportsSelectDistinct();
supportsSelectOrdering = commandCapabilities->
    SupportsSelectOrdering();
supportsSelectGrouping = commandCapabilities->
    SupportsSelectGrouping();
```


References

For more information, see these *FDO API Reference Help* topics:

- class FdoICommandCapabilities
- enum FdoCommandType
- enum FdoRdbmsCommandType

Expression Capabilities

Code

Declarations

The object and variable declarations are the following:

```
FdoIExpressionCapabilities * expressionCapabilities;  
FdoInt32 numExpressionTypes = 0;  
// this is an enumerated type  
FdoExpressionType * expressionTypes;  
FdoInt32 numFunctionDefinitions = 0;  
FdoFunctionDefinitionCollection * functions;  
FdoFunctionDefinition * functionDefinition;  
FdoString * functionName;  
FdoString * functionDescription;  
FdoInt32 numArgumentDefinitions = 0;  
FdoReadOnlyArgumentDefinitionCollection * arguments;  
FdoArgumentDefinition * argumentDefinition;  
FdoString * argumentName;  
FdoString * argumentDescription;  
FdoDataType argumentType;
```

Method Calls

The method calls are the following:

```

expressionCapabilities = connection->GetExpressionCapabilities();
// this returns a list of expression types
expressionTypes = expressionCapabilities->GetExpressionTypes();
// loop through the expression Types
functions = expressionCapabilities->GetFunctions();
numFunctionDefinitions = functions->GetCount();
for (int i = 0; i < numFunctionDefinitions; i++) {
    functionDefinition = functions->GetItem(i);
    functionName = functionDefinition->GetName();
    functionDescription = functionDefinition->GetDescription();
    arguments = functionDefinition->GetArguments();
    numArgumentDefinitions = arguments->GetCount();
    for (int j = 0; j < numArgumentDefinitions; j++) {
        argumentDefinition = arguments->GetItem(j);
        argumentName = argumentDefinition->GetName();
        argumentDescription = argumentDefinition->GetDescription();
        argumentType = argumentDefinition->GetDataType();
    }
}

```

Filter Capabilities

Code

Declarations

The object and variable declarations are the following:

```

FdoIFilterCapabilities * filterCapabilities;
FdoInt32 numConditionTypes = 0;
FdoConditionType * conditionTypes;
FdoInt32 numSpatialOperations = 0;
FdoSpatialOperations * spatialOperations;
FdoInt32 numDistanceOperations = 0;
FdoDistanceOperations * distanceOperations;
bool supportsGeodesicDistance;
bool supportsNonLiteralGeometricOperations;

```

Method Calls

The method calls are the following:

```

filterCapabilities = connection->GetFilterCapabilities();
conditionTypes = filterCapabilities->
    GetConditionTypes(numConditionTypes);
// loop through conditionTypes
spatialOperations = filterCapabilities->
    GetSpatialOperations(numSpatialOperations);
// loop through spatialOperations
distanceOperations = filterCapabilities->
    GetDistanceOperations(numSpatialOperations);
// loop through distanceOperations
supportsGeodesicDistance = filterCapabilities->
    SupportsGeodesicDistance();
supportsNonLiteralGeometricOperations = filterCapabilities->
    SupportsNonLiteralGeometricOperations();

```

Geometry Capabilities

Code

Declarations

The object and variable declarations are the following:

```

FdoIGeometryCapabilities * geometryCapabilities;
FdoInt32 numGeometryTypes = 0;
FdoGeometryType * geometryTypes;
FdoInt32 numGeometryComponentTypes = 0;
FdoGeometryComponentType * geometryComponentTypes;
FdoInt32 dimensionalities = 0;

```

Method calls

The method calls are the following:

```

geometryCapabilities = connection->GetGeometryCapabilities();
geometryTypes = geometryCapabilities->
    GetGeometryTypes(numGeometryTypes);
// loop through geometryTypes
geometryComponentTypes = geometryCapabilities->
    GetGeometryComponentTypes(numGeometryComponentTypes);
// loop through geometryComponentTypes
dimensionalities = geometryCapabilities->GetDimensionalities();
// FdoDimensionality_XY is 0 and so is always a given
if (dimensionalities & FdoDimensionality_Z) {
    // do whatever
}
if (dimensionalities & FdoDimensionality_M) {
    // do whatever
}

```

Raster Capabilities

Code

Declarations

The object and variable declarations are the following:

```

FdoIRasterCapabilities * rasterCapabilities;
bool supportsRaster;
bool supportsStitching;
bool supportsSubsampling;
bool supportsDataModel;
FdoRasterDataModel * rgbRasterDataModel;

```

Method calls

The method calls are the following:

```

rasterCapabilities = connection->GetRasterCapabilities();
supportsRaster = rasterCapabilities->SupportsRaster();
if (supportsRaster) {
    supportsStitching = rasterCapabilities->SupportsStitching();
    supportsSubsampling = rasterCapabilities->SupportsSubsampling();
    rgbRasterDataModel = FdoRasterDataModel::Create();
    rgbRasterDataModel->
        SetDataModelType(FdoRasterDataModelType_RGB);
    rgbRasterDataModel->SetBitsPerPixel(64);
    rgbRasterDataModel->
        SetOrganization(FdoRasterDataOrganization_Image);
    rgbRasterDataModel->SetTileSizeX(64);
    rgbRasterDataModel->SetTileSizeY(128);
    supportsDataModel = rasterCapabilities->
        SupportsDataModel(rgbRasterDataModel);
}

```


Schema Management

7

This chapter describes how to create and work with schemas and explains some issues related to schema management. For example, you can use the FDO feature schema to specify how to represent geospatial features.

Schema Package

The FDO feature schema provides a logical mechanism for specifying how to represent geospatial features. FDO providers are responsible for mapping the feature schema to some underlying physical data store. The FDO feature schema is based somewhat on a subset of the OpenGIS and ISO feature models. It supports both non-spatial features and spatial features.

The Schema package contains a collection of classes that define the logical feature schema. These classes can be used to set up a feature schema and to interrogate the metadata from a provider using an object-oriented structure. The logical feature schema provides a logical view of geospatial feature data that is fully independent from the underlying storage schema. All data operations in FDO are performed against the classes and relationships defined by the logical feature schema. For example, different class types in the feature schema are used to describe different types of geospatial objects and spatial features.

Base Properties

All classes in the feature schema support the concept of base properties, which are properties that are pre-defined either by the FDO API or by a specific FDO feature provider. For example, all classes in the schema have two base properties: `ClassName` and `SchemaName`. These properties can be used to query across an inheritance hierarchy or to process the results of heterogeneous queries. FDO

feature providers can also predefine base properties. The following base properties are predefined by the FDO API:

Property Name	Required	Description
SchemaName	Y	Name of the schema to which objects of the class belong; read-only string.
ClassName	Y	Name of the class that defines the object; read-only string.
RevisionNumber	N	Revision number of the object; read-only 64-bit integer. NOTE Some providers may use this property to support optimistic locking.

Cross-Schema References

Some FDO feature providers may support multiple schemas. For these providers, the feature schema supports the concept of cross-schema references for classes. This means that a class in one schema may derive from a class in another schema, relate to a class in another schema, or contain an object property definition that is based on a class in another schema.

Parenting in the Schema Classes

The feature schema object model defined in the FDO API supports full navigation through parenting. That is, once a schema element is added to an `FdoFeatureSchema` class, it can navigate the object hierarchy upward to the root `FdoFeatureSchema` and, from there, to any other element in the feature schema. This parenting support is fully defined in the `FdoSchemaElement` abstract base class.

When inserting features that have object collections, the parent object instance must be identified when inserting the child objects (for example, a parent class “Road” has an object property called “sidewalks” of type “Sidewalk”). For more information, see [Data Maintenance](#) on page 79.

Physical Mappings

Each feature provider maps the logical feature schema to an underlying physical data store. Some feature providers may provide some level of control

over how the logical schema gets mapped to the underlying physical storage. For example, an RDBMS-based feature provider may allow table and column names to be specified for classes and properties. Since this is entirely provider-dependent, the FDO API simply provides abstract classes for passing physical schema and class mappings to the provider (`FdoPhysicalSchemaMapping`, `FdoPhysicalClassMapping`, `FdoPhysicalPropertyMapping`, and `FdoPhysicalElementMapping`, respectively). The implementation of these abstract classes is up to each feature provider.

Schema Overrides

Using schema overrides, FDO applications can customize the mappings between Feature (logical) Schemas and the Physical Schema of the provider data store.

Schema overrides are provider-specific because different FDO providers support FDO data stores with widely different physical formats. Therefore, the types of schema mappings in these overrides also vary between providers. For example, an RDBMS-type provider might provide a mapping to index a set of columns in a class table. However, other providers would not necessarily be able to work with the concept of an index. For information about schema overrides support by a specific provider, see the appropriate appendix in this document and *The Essential FDO*.

NOTE Some providers support only default schema mappings.

Working with Schemas

There are three primary operations involved with schema management:

- Creating a schema
- Describing a schema
- Modifying a schema

Creating a Schema

The following basic steps are required to create a schema (some steps are optional; some may be done in an alternate order to achieve the same result):

- Use the `FdoFeatureSchema::Create("SchemaName", "FeatureSchema Description")` method to create a schema.
- Use the `FdoFeatureSchema::GetClasses()` method to return a class collection.
- Use the `FdoClass::Create("className", "classDescription")` or `FdoFeatureClass::Create("className", "classDescription")` method to create `FdoClass` or `FdoFeatureClass` type objects.
- Use the `FdoClassCollection::Add(class)` method to add `FdoClass` or `FdoFeatureClass` objects to the class collection.
- Use the `FdoGeometricPropertyDefinition::Create("name", "Description")` method to create `FdoGeometryProperty`.
- Use the `FdoDataPropertyDefinition::Create("name", "Description")` method to create `FdoDataProperty`.
- Use the `FdoObjectPropertyDefinition::Create("name", "Description")` method to create `FdoObjectProperty`.
- Use the `FdoClassDefinition::GetProperties()` and `Add(property)` methods to add property to class definition.
- Use the `FdoIApplySchemaCommand::SetFeatureSchema(feature schema)` method to set the schema object for the `IFdoIApplySchemaCommand`.
- Use the `FdoAssociationPropertydefinition` class to represent the association between two classes. The class of the associated class must already be defined in the feature schema and cannot be abstract.
- Use the `FdoIApplySchemaCommand::Execute()` method to execute changes to the feature schema.

For an example of schema creation, see [Example: Creating a Feature Schema](#) on page 72.

Use the `FdoClassDefinition::GetIdentityProperties()` and `Add(Property Object)` methods to set the property as `FdoClass` or `FdoFeatureClass` Identifier. FDO allows multiple Identifiers for both types of classes, although Identifiers have slight differences in both cases.

FDOFeatureClass

FdoFeatureClass is a class that defines features. In the case of GIS, they would often be spatial features, having some sort of geometry associated with them. In most providers, FdoFeatureClass requires a unique identifier to distinguish the features.

However, there are identifiers only if no base class exists. If the base class has an identifier, the child class does not have one. You cannot set an identifier to the child class. Any class definition that has a base class cannot also have any identity properties because it inherits from the base class.

Therefore, you cannot send an identifier when a feature class is a child since it always inherits the identifier from the base class.

FDOClass

This class is used for non-spatial data. It can act as a stand-alone class, where it would have no association with any other class, or if the FdoClass is being used as an ObjectProperty, it can be used to define properties of some other FdoClass or FdoFeatureClass.

ObjectProperty Types

ObjectProperties have the following types:

- Value
- Collection
- OrderedCollection

The Value ObjectProperty type has a relationship of one-to-one, providing a single value for each property.

The Collection and OrderedCollection ObjectProperty types have a one-to-many relationship, where many ObjectProperties may be associated with one property. Ordered Collections can be stored in an ascending or descending order

At least one Identifier will be required if the FdoClass is to be used as a stand-alone Class.

- All Identifiers for FdoDataType_Int64 must not be Read-Only, since none of these will be an auto-generated property value.
- If creating multiple Identifiers, all Identifiers must be set to NOT NULL.

Non-Feature Class Issues

A non-feature class in FDO can be created as a stand-alone class, a contained class, or both. As a contained class, it defines a property of another class or feature class (see FdoFeatureClass and FdoClassType Global Enum). How this non-feature class is created affects the way the data is inserted, queried, and updated.

Stand-alone Class

This type of class stores non-feature data (for example, manufacturers). The FdoClassType_Class must be created with one or more identity properties (see FdoObjectPropertyDefinition), which is required in order that the class has a physical container (that is, a table in the RDBMS) associated with it. If the class is created without specifying an IdentityProperty, only the definition is stored in the metadata, which prevents any direct data inserts.

Contained Class

This type of class stores non-feature data that defines a property of another class or feature class (for example, Sidewalk could be a property of a Road feature class; the Sidewalk class defines the Road.Sidewalk property). In this case, the FdoClassType_Class does not need to be created with one or more identity properties, although it can be.

Class With IdentityProperty Used as ObjectProperty

This type of class reacts like a stand-alone class; however, with this type, it is possible to do direct data inserts. It can also be populated through a container class (for example, Road.Sidewalk) since it defines an object property (see FdoObjectPropertyDefinition). If this class is queried directly, only the data inserted into the class as a stand-alone is returned. The data associated with the ObjectProperty can only be queried through the container class (for example, Road.Sidewalk).

Class Without a Defined IdentityProperty Used as ObjectProperty

Because this class has no defined IdentityProperty, it can only be populated through the container class (for example, Road.Sidewalk) since it defines ObjectProperty. This class cannot be queried directly. The data associated with the object property can only be queried through the container class (for example, Road.Sidewalk). As an object property, it is defined as one of the following:

- **Value type property.** Does not need any identifier since it has a one-to-one relationship with the container class.
- **Collection type property.** Requires a local identifier, which is an identifier defined when creating the ObjectProperty object.
- **Ordered Collection type property.** Requires a local identifier, which is an identifier defined when creating the ObjectProperty object.

When defining either a Collection or Ordered Collection type ObjectProperty, you must set an IdentityProperty attribute for that object property. This ObjectClass.IdentityProperty acts only as a local identifier compared to the IdentityProperty set at the class level. As a local identifier, it acts to uniquely identify each item within each collection (for example, if the local identifier for Road.Sidewalk is Side, there can be multiple sidewalks with Side="Left" but only one per Road).

Describing a Schema

Use the FdoDescribeSchema::Execute function to retrieve an FdoFeatureSchemaCollection in order to obtain any information about existing schema(s). The FdoFeatureSchemaCollection consists of all FdoFeatureSchemas in the data store and can be used to obtain information about any schema object, including FdoFeatureSchema, FdoClass, FdoFeatureClass, and their respective properties. The following functions return the main collections required to obtain information about all schema objects:

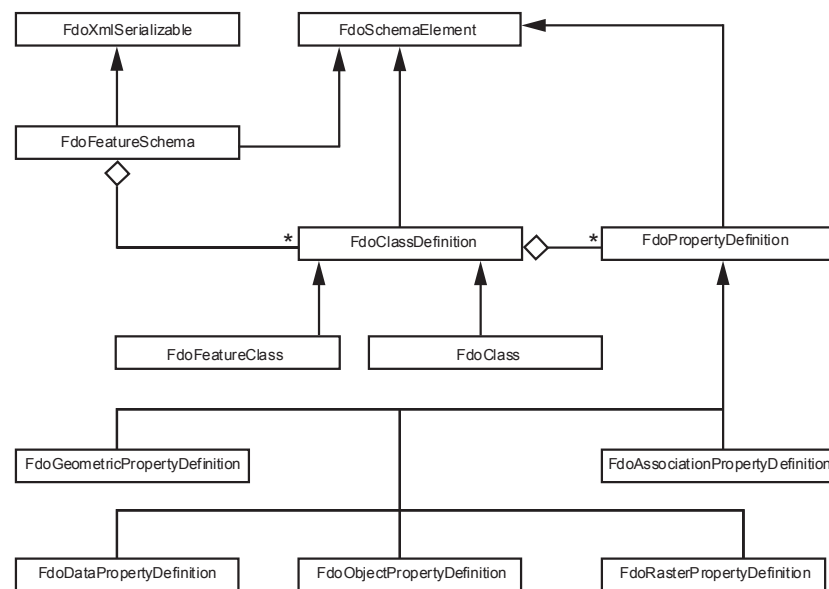
- FdoFeatureSchema::GetClasses method obtains FdoClass and FdoFeatureClasses.
- FdoClassDefinition::GetProperties method obtains a FdoPropertyDefinitionCollection.
- FdoClassDefinition::GetBaseProperties method obtains a FdoPropertyDefinitionCollection of the properties inherited from the base classes.

NOTE Even if your schema has no base classes (inheritance), all classes will inherit some properties from system classes.

Use these functions throughout the application to obtain any information about schema objects. For example, in order to insert data into a class, you must use these functions to determine what data type is required. Description of the data is separate from actions.

The example in the following link is a simple function that shows how to use `FdoIDescribeSchema` and loop through the schema and class containers to search for duplicate class names. It searches all schemas to ensure that the class name does not exist in any schema in the data store. Class names must be unique across the entire FDO database.

For a schema description example, see [Example: Describing a Schema and Writing It to an XML File](#) on page 75 NO LABEL .



FDO Schema Element Class Diagram

Modifying Models

Add schema elements to a model by inserting them into the appropriate collection.

Elements are removed from the model by using either of the following methods:

- Call the `FdoSchemaElement::Delete()` method. This flags the element for deletion when the changes are accepted (generally through `FdoIApplySchema`), but the element remains a member of all collections until that time.
- Remove the element from the appropriate collection via the `FdoSchemaCollection::Remove()` or `FdoSchemaCollection::RemoveAt()` methods. This immediately disassociates the element from the collection.

Schema Element States

All elements within the model maintain a state flag. This flag can be retrieved by calling `FdoSchemaElement::GetElementState()`, but it cannot be directly set. Instead, its state changes in reaction to the changes made to the model:

- **Unchanged.** When a schema model is retrieved via `FdoIDescribeSchema`, all elements are initially marked Unchanged.
- **Detached.** Removing an element from an owning collection sets its state to Detached.
- **Deleted.** Calling the `Delete()` method on an element sets its state to Deleted.
- **Added.** Placing an element within a collection marks the element as Added.
- **Modified.** When adding or removing a sub-element, such as a property element from a class, the class element state will be changed to Modified.

Additionally, when an element that is contained by another element is changed in any way, the containing element is also marked as Modified. So, for example, if a new value is added to the `SchemaAttributeDictionary` of the “Class3” element in our model, both the “Class3” `FdoClass` object and the `FdoFeatureSchema` object would be marked as Modified.

The state flags are maintained until the changes are accepted, that is, when `IApplySchema` is executed. At that time, all elements marked `Deleted` are released and all other elements are set to `Unchanged`.

NOTE When you remove an element from an owning collection, its state is marked as `Detached`. All collections currently in FDO are owning collections, except for one, the collections `FdoClassDefinition::GetIdentityProperties()`.

Rollback Mechanism

The `FdoFeatureSchema` contains a mechanism that allows you to “roll back” model changes to the last accepted state. For example, a model retrieved via `FdoDescribeSchema` can have classes added, attributes deleted, or names and default values changed. All of these changes are thrown out and the model returned to its unmodified state by calling `FdoFeatureSchema::RejectChanges()`.

The converse of this operation is the `FdoFeatureSchema::AcceptChanges()` method, which removes all of the elements with a status of `Deleted` and sets the state flag of all other elements to `Unchanged`. Generally, this method is only invoked by FDO provider code after it has processed an `FdoApplySchema::Execute()` command. Normal FDO clients should not call this method directly.

FDO XML Format

FDO feature schemas can be written to an XML file. The `FdoFeatureSchema` and `FdoFeatureSchemaCollection` classes support the `FdoXmlSerializable` interface. The sample code shows an `FdoFeatureSchema` object calling the `WriteXml()` method to generate an XML file containing the feature schema created by the sample code.

FDO feature schemas can also be read from an XML file. The `FdoFeatureSchemaCollection` class supports the `FdoXmlDeserializable` interface. The sample code shows an `FdoFeatureSchemaCollection` object calling the `ReadXml()` method to read a set of feature schemas into memory from an XML file. The code shows the desired schema being retrieved from the collection and applied to the data store.

The XML format used by FDO is a subset of the Geography Markup Language (GML) standardized by the Open GIS Consortium (OGC). One thing shown in the sample code is a round-trip conversion from FDO feature schema to GML schema back to FDO feature schema. To accomplish this round-trip, the

ReadXml() method supports a superset of the GML that is written by the WriteXml() method.

The following table specifies the mapping of FDO feature schema elements to GML elements and attributes. This mapping is sufficient to understand the XML file generated from the schema defined by the sample code. It also provides a guide for writing a GML schema file by hand. This file can then be read in and applied to a data store. For more information, see [Example: Creating a Schema Read In from an XML File](#) on page 75.

Another form of round-trip translation would be from a GML schema produced by another vendor's tool to an FDO feature schema, and then back to a GML schema. However, the resemblance the of resulting GML schema to the original GML schema might vary from only roughly equivalent to being exactly the same.

Map FDO Element to GML Schema Fragment

FDO Element	GML Schema Fragment
FeatureSchema	<pre> <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" targetNamespace="http://<customer_url>/<FeatureSchemaName>" xmlns:fdo="http://fdo.osgeo.org/isd/schema" xmlns:gml="http://www.opengis.net/gml" xmlns:<FeatureSchemaName>="http://<customer_url>/<FeatureSchemaName>" elementFormDefault="qualified" attributeFormDefault="unqualified" > { see <MetaData> } { optional xs:import element to enable schema validation <xs:import namespace="http://fdo.osgeo.org/schema" schemaLocation="<FDO SDK Install Location>/docs/XmlSchema/FdoDocument.xsd"/> } { <one xs:element and/or xs:complexType per class> } </xs:schema> </pre>

FDO Element	GML Schema Fragment
ClassDefinition (with identity properties)	<pre> <xs:element name="<className>" type="<className>Type" abstract="<true false>" substitutionGroup="gml:_Feature" > <xs:key name="<className>Key"> <xs:selector xpath="."/.></xs:selector> <xs:field xpath="<identityPropertyName>" /> <xs:field xpath="..." /> <xs:field xpath="<identityProperty{n}>Name" /> </xs:key> </xs:element> </pre>
FeatureClass	<pre> <xs:element ...see ClassDefinition (with identity properties)...></xs:element> <xs:complexType name="<className>Type" abstract="<true false>" /> { see FeatureClass.GeometryProperty } > { see <MetaData> } <xs:complexContent> <xs:extension base="{baseClass} ? {baseClass.schema.name}:{baseClass.name} : 'gml:AbstractFeatureType' " > <xs:sequence> { list of properties; see DataProperty, Geometric Property } </xs:sequence> </xs:extension> </xs:complexContent> </xs:complexType> </pre>

FDO Element	GML Schema Fragment
FeatureClass. GeometryProperty	<pre> <!-- these attributes belong to the xs:complexType element --> fdo:geometryName="<geometryPropertyName>" fdo:geometricTypes="<list of FdoGeometricTypes>" fdo:geometryReadOnly="<true false>" fdo:hasMeasure="<true false>" fdo:hasElevation="<true false>" fdo:srsName="<spatialContextName>"/> </pre>
DataProperty (decimal or string)	<pre> <!-- minOccurs attribute generated only if value is 1 default attribute generated only if a default value exists fdo:readOnly attribute generated only if value is true --> <xs:element name="<propertyName>" minOccurs="{isNullable ? 0 : 1}" default="<defaultValue>" fdo:readOnly="<true false>" > { see <MetaData> } <xs:simpleType> { see DataType String or DataType Decimal } </xs:simpleType> </xs:element> </pre>
DataProperty (other type)	<pre> <xs:element name="<propertyName>" type="<datatype>" minOccurs="{isNullable ? 0 : 1}" default="<defaultValue>" fdo:readOnly="<true false>" > { see <MetaData> } </xs:element> </pre>
DataType String	<pre> <xs:restriction base="xs:string"> <xs:maxLength value="<length>"/> </xs:restriction> </pre>

FDO Element	GML Schema Fragment
DataType Decimal	<pre> <xs:restriction base="xs:decimal"> <xs:totalDigits value="<precision>" /> <xs:fractionDigits value="<scale>" /> </xs:restriction> </pre>
GeometricProperty (not a defining FeatureClass GeometryProperty)	<pre> <xs:element name="<propertyName>" type="gml:AbstractGeometryType" fdo:geometryName="<propertyName>" fdo:geometricTypes="<list of FdoGeometricTypes>" fdo:geometryReadOnly="<true false>" fdo:hasMeasure="<true false>" fdo:hasElevation="<true false>" fdo:srsName="<spatialContextName>" /> > { see <MetaData> } </xs:element> </pre>

FDO Element	GML Schema Fragment
MetaData	<pre> <!-- the pattern referenced in the xs:schema element for FeatureSchema--> <xs:annotation> <xs:documentation>{description arg to static FdoFeaturesS chema::Create() }</xs:documentation> </xs:annotation> <!-- the pattern referenced in the xs:element element for DataProperty --> <xs:annotation> <xs:documentation>{description arg to static FdoDataProp ertyDefinition::Create() }</xs:documentation> </xs:annotation> <!-- the pattern referenced in the xs:element element for a non-feature-defining GeometricProperty --> <xs:annotation> <xs:documentation>{description arg to static FdoGeomet ricPropertyDefinition::Create() }</xs:documentation> </xs:annotation> <!-- the pattern referenced in the xs:complexType element for FeatureClass --> <xs:annotation> <xs:documentation>{description arg to static FdoFeature Class::Create() }</xs:documentation> <xs:appinfo source="<uri>" /> <xs:documentation>{description arg to static FdoGeomet ricPropertyDefinition::Create() }</xs:documentation> </xs:annotation> </pre>

Map FDO Datatype to GML Type	
FDO Datatype	GML Type
Boolean	xs:boolean
Byte	fdo:Byte
DateTime	xs:dateTime

FDO Datatype	GML Type
Double	xs:double
Int16	fdo:Int16
Int32	fdo:Int32
Int64	fdo:Int64
Single	xs:float
BLOB	xs:base64Binary
CLOB	xs:string

Creating and Editing a GML Schema File

The sample in this section illustrates the creation of a GML schema file containing the definition of an FDO feature schema that contains one feature. The name of this file will have the standard XML schema extension name, *.xsd*. This means that it contains only one schema and that the root element is `xs:schema`. The `ReadXml()` method will take a filename argument whose extension is either *.xsd* or *.xml*. In the latter case, the file could contain many schema definitions. If it does, each schema is contained in an `xs:schema` element, and all `xs:schema` elements are contained in the `fdo:DataStore` element. If there is only one schema in the *.xml* file, then the `fdo:DataStore` element is not used, and the root element is `xs:schema`.

You may want to validate the schema that you create. To do so, you must include the optional `xs:import` line specified in the GML schema fragment for `FeatureSchema`.

The sample feature implements a table definition for the Buildings feature in the Open GIS Consortium document 98-046r1. This table definition is expressed in an XML format on page 14 of the document and is reproduced as follows:

```

<ogc-sfsql-table>
  <table-definition>
    <name>buildings</name>
    <column-definition>
      <name>fid</name>
      <type>INTEGER</type>
      <constraint>NOT NULL</constraint>
      <constraint>PRIMARY KEY</constraint>
    </column-definition>
    <column-definition>
      <name>address</name>
      <type>VARCHAR(64)</type>
    </column-definition>
    <column-definition>
      <name>position</name>
      <type>POINT</type>
    </column-definition>
    <column-definition>
      <name>footprint</name>
      <type>POLYGON</type>
    </column-definition>
  </table-definition>

```

Add GML for the FDO Feature Schema

Start with the skeleton GML for an FDO Feature Schema with the <MetaData> reference replaced by the valid pattern:

```

<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://<customer_url>/<FeatureSchemaName>"
  xmlns:fdo="http://fdo.osgeo.org/schema"
  xmlns:gml="http://www.opengis.net/gml"
  xmlns:<FeatureSchemaName>="http://<customer_url>/<FeatureSchemaName>"
  elementFormDefault="qualified"
  attributeFormDefault="unqualified"
>
  <xs:annotation>
    <xs:documentation>
      {description arg to static FdoFeatureSchema::Create()}
    </xs:documentation>
  </xs:annotation>
  { <one xs:element and/or xs:complexType per class> }
</xs:schema>

```

For <customer_url> substitute “fdo_customer”. For <FeatureSchemaName> substitute “OGC980461FS”, and for {description arg ... } substitute “OGC Simple Features Specification for SQL.”

Add GML for an FDO Feature Class

Start with the GML that is already written and add the skeleton for an FDO Feature Class, which includes the skeleton for a class definition with identity properties. The <MetaData> is replaced with the valid pattern.


```

<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://fdo_customer/OGC980461FS"
  xmlns:fdo="http://fdo.osgeo.org/schema"
  xmlns:gml="http://www.opengis.net/gml"
  xmlns:OGC980461FS="http://fdo_customer/OGC980461FS"
  elementFormDefault="qualified"
  attributeFormDefault="unqualified"
>
  <xs:annotation>
    <xs:documentation>OGC Simple Features Specification for
      SQL</xs:documentation>
  </xs:annotation>
  <xs:element name="<className>"
    type="<className>Type"
    abstract="<true | false>"
    substitutionGroup="gml:_Feature"
  >
    <xs:key name="<className>Key">
      <xs:selector xpath="."/.></xs:selector>
      <xs:field xpath="<identityPropertyName>"></xs:field>
    </xs:key>
  </xs:element>
  <xs:complexType name="<className>Type"
    abstract="<true | false>">
    fdo:geometryName="<geometryPropertyName>"
    fdo:geometricTypes="<list of FdoGeometricTypes>"
    fdo:geometryReadOnly="<true | false>"
    fdo:hasMeasure="<true | false>"
    fdo:hasElevation="<true | false>"
    fdo:srsName="<spatialContextName>">
  >
    <xs:annotation>
      <xs:documentation>{description arg to static
        FdoFeatureClass::Create()}</xs:documentation>
      <xs:appinfo source="<uri>"></xs:appinfo>
      <xs:documentation>{description arg to static
        FdoGeometricPropertyDefinition::Create()}
      </xs:documentation>
    </xs:annotation>
    <xs:complexContent>
      <xs:extension base="{baseClass} ?
        {baseClass.schema.name}:{baseClass.name} :
        'gml:AbstractFeatureType' "

```

```

>
<xs:sequence>
  { list of properties; see DataProperty, GeometricProperty
}
</xs:sequence>
</xs:extension>
</xs:complexContent>
</xs:complexType>
</xs:schema>

```

You can make the following changes:

- For <className> substitute “buildings”.
- Set the value of the xs:element abstract attribute to false.
- For <identityPropertyName> substitute “fid”. A data property whose name is “fid” will be added.
- Set the value of the xs:complexType abstract attribute to false.
- For <geometryPropertyName> substitute “footprint”.
- For <list of FdoGeometricTypes> substitute “surface”.
- Set the values of fdo:geometryReadOnly, fdo:hasMeasure, and fdo:hasElevation to false.
- For <spatialContextName> substitute “SC_0”.
- For {description arg to FdoFeatureClass::Create()} substitute “OGC 98-046r1 buildings”.
- For <uri> substitute “http://fdo.osgeo.org/schema”.
- For {description arg to FdoGeometricPropertyDefinition::Create()} substitute “a polygon defines a building perimeter”.
- This class has no base class so set the value of the xs:extension base attribute to ‘gml:AbstractFeatureType’.

Add GML for Property Definitions

An integer data property whose name is “fid” will be added. This property is already identified as an identity property in the xs:key element. A string data property whose name is “name” and a geometry property whose name is “position” will also be added.

```

<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://fdo_customer/OGC980461FS"
  xmlns:fdo="http://fdo.osgeo.org/schema"
  xmlns:gml="http://www.opengis.net/gml"
  xmlns:OGC980461FS="http://fdo_customer/OGC980461FS"
  elementFormDefault="qualified"
  attributeFormDefault="unqualified"
>
  <xs:annotation>
    <xs:documentation>OGC Simple Features Specification for
      SQL</xs:documentation>
  </xs:annotation>
  <xs:element name="buildings"
    type="buildingsType"
    abstract="false"
    substitutionGroup="gml:_Feature"
  >
    <xs:key name="buildingsKey">
      <xs:selector xpath="."/></buildings"/>
      <xs:field xpath="fid"/>
    </xs:key>
  </xs:element>
  <xs:complexType name="buildingsType"
    abstract="false">
    fdo:geometryName="footprint"
    fdo:geometricTypes="surface"
    fdo:geometryReadOnly="false"
    fdo:hasMeasure="false"
    fdo:hasElevation="false"
    fdo:srsName="SC_0"/>
  >
    <xs:annotation>
      <xs:documentation>OGC 98-046r1 buildings
      </xs:documentation>
      <xs:appinfo source="http://fdo.osgeo.org/schema"/>
      <xs:documentation>a polygon defines the perimeter of a
        building</xs:documentation>
    </xs:annotation>
    <xs:complexContent>
      <xs:extension base="gml:AbstractFeatureType"
      >
        <xs:sequence>
          <xs:element name="<propertyName>"

```

```

        type="<datatype>"
        minOccurs="{isNullable ? 0 : 1}"
        default="<defaultValue>"
        fdo:readOnly="<true | false>"
    >
        <xs:annotation>
            <xs:documentation>{description arg to static
                FdoDataPropertyDefinition::Create() }
            </xs:documentation>
        </xs:annotation>
    </xs:element>
    <xs:element name="<propertyName>"
        minOccurs="{isNullable ? 0 : 1}"
        default="<defaultValue>"
        fdo:readOnly="<true | false>"
    >
        <xs:annotation>
            <xs:documentation>{description arg to static
                FdoDataPropertyDefinition::Create() }
            </xs:documentation>
        </xs:annotation>
        <xs:simpleType>
            <xs:restriction base="xs:string">
                <xs:maxLength value="<length>"/>
            </xs:restriction>
        </xs:simpleType>
    </xs:element>
    <xs:element name="<propertyName>"
        ref="gml:_Geometry"
        fdo:geometryName="<propertyName>"
        fdo:geometricTypes="<list of FdoGeometricTypes>"
        fdo:geometryReadOnly="<true | false>"
        fdo:hasMeasure="<true | false>"
        fdo:hasElevation="<true | false>"
        fdo:srsName="<spatialContextName>"/>
    >
        <xs:annotation>
            <xs:documentation>{description arg to static
                FdoGeometricPropertyDefinition::Create() }
            </xs:documentation>
        </xs:annotation>
    </xs:element>
</xs:sequence>

```

```

        </xs:extension>
    </xs:complexContent>
</xs:complexType>
</xs:schema>

```

You can make the following changes:

- For the first data property <propertyName> substitute “fid”.
- For the first data property <dataType> substitute “fdo:int32”.
- Do not include the minOccurs or default attributes because the value of minOccurs is 0, which is the default, and there is no <defaultValue>.
- Set the fdo:readOnly attribute for “fid” to false.
- Set the content for xs:documentation for “fid” to “feature id”.
- For the second data property <propertyName> substitute “address”.
- Do not include the minOccurs or default attributes because the value of minOccurs is 0, which is the default, and there is no <defaultValue>.
- Set the fdo:readOnly attribute for “name” to false.
- Set the content for xs:documentation for “address” to “address of the building”.
- For <length> substitute “64”.
- For the geometry property <propertyName> substitute “position”.
- For <list of FdoGeometricTypes> substitute “point”.
- Set the values of fdo:geometryReadOnly, fdo:hasMeasure, and fdo:hasElevation to false.
- For <spatialContextName> substitute “SC_0”.
- For {description arg to FdoGeometricPropertyDefinition::Create()} substitute “position of the building”.

The Final Result

After all the required substitutions, the GML for the schema containing the Buildings feature is as follows:

```

<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://fdo_customer/OGC980461FS"
  xmlns:fdo="http://fdo.osgeo.org/schema"
  xmlns:gml="http://www.opengis.net/gml"
  xmlns:OGC980461FS="http://fdo_customer/OGC980461FS"
  elementFormDefault="qualified"
  attributeFormDefault="unqualified"
>
  <xs:annotation>
    <xs:documentation>OGC Simple Features Specification for
      SQL</xs:documentation>
  </xs:annotation>
  <xs:element name="buildings"
    type="buildingsType"
    abstract="false"
    substitutionGroup="gml:_Feature"
  >
    <xs:key name="buildingsKey">
      <xs:selector xpath="//buildings"/>
      <xs:field xpath="fid"/>
    </xs:key>
  </xs:element>
  <xs:complexType name="buildingsType"
    abstract="false"/>
    fdo:geometryName="footprint"
    fdo:geometricTypes="surface"
    fdo:geometryReadOnly="false"
    fdo:hasMeasure="false"
    fdo:hasElevation="false"
    fdo:srsName="SC_0"/>
  >
    <xs:annotation>
      <xs:documentation>OGC 98-046r1 buildings
      </xs:documentation>
      <xs:appinfo source="http://fdo.osgeo.org/schema"/>
      <xs:documentation>a polygon defines the perimeter of a
        building</xs:documentation>
    </xs:annotation>
    <xs:complexContent>
      <xs:extension base="gml:AbstractFeatureType"
      >
        <xs:sequence>
          <xs:element name="fid"

```

```

        type="fdo:int32"
        fdo:readOnly="false"
    >
        <xs:annotation>
            <xs:documentation>feature id
            </xs:documentation>
        </xs:annotation>
    </xs:element>
    <xs:element name="address"
        fdo:readOnly="false"
    >
        <xs:annotation>
            <xs:documentation>address of the building
            </xs:documentation>
        </xs:annotation>
        <xs:simpleType>
            <xs:restriction base="xs:string">
                <xs:maxLength value="64"/>
            </xs:restriction>
        </xs:simpleType>
    </xs:element>
    <xs:element name="position"
        ref="gml:_Geometry"
        fdo:geometryName="position"
        fdo:geometricTypes="point"
        fdo:geometryReadOnly="false"
        fdo:hasMeasure="false"
        fdo:hasElevation="false"
        fdo:srsName="SC_0"/>
    >
        <xs:annotation>
            <xs:documentation>position of the building</xs:docu
mentation>
        </xs:annotation>
    </xs:element>
</xs:sequence>
</xs:extension>
</xs:complexContent>
</xs:complexType>
</xs:schema>

```

Schema Management Examples

Example: Creating a Feature Schema

The following sample code creates an `FdoFeatureSchema` object called “SampleFeatureSchema.” The schema contains one class, which has three properties. The class and its properties conform to the table definition for the Lakes feature in the Open GIS Consortium document 98-046r1. This table definition is expressed in an XML format on page 10 of the document and is reproduced as follows:

```
<ogc-sfsql-table>
  <table-definition>
    <name>lakes</name>
    <column-definition>
      <name>fid</name>
      <type>INTEGER</type>
      <constgraint>NOT NULL</constraint>
      <constraint>PRIMARY KEY<constraint>
    </column-definition>
    <column-definition>
      <name>name</name>
      <type>VARCHAR(64)</type>
    </column-definition>
    <column-definition>
      <name>shore</name>
      <type>POLYGON</type>
    </column-definition>
  </table-definition>
```

The table definition whose name is “lakes” is mapped to an `FdoFeatureClass` object called “SampleFeatureClass.” The column definition whose name is “fid” is mapped to an `FdoDataPropertyDefinition` object called “SampleIdentityDataProperty.” The column definition whose name is “name” is mapped to an `FdoDataPropertyDefinition` object called “SampleNameDataProperty.” The column definition whose name is “shore” is mapped to an `FdoGeometricPropertyDefinition` object called “SampleGeometricProperty.”


```

// Create the ApplySchema command
FdoPtr<FdoIApplySchema> sampleApplySchema;
sampleApplySchema = (FdoIApplySchema *)
    connection->CreateCommand(FdoCommandType_ApplySchema);
// Create the feature schema
FdoPtr<FdoFeatureSchema> sampleFeatureSchema;
sampleFeatureSchema = FdoFeatureSchema::Create(L"SampleFeaturesS
chema", L"Sample Feature Schema Description");
// get a pointer to the feature schema's class collection
// this object is used to add classes to the schema
FdoPtr<FdoClassCollection> sampleClassCollection;
sampleClassCollection = sampleFeatureSchema->GetClasses();
// create a feature class, i.e., a class containing a geometric
// property set some class level properties
FdoPtr<FdoFeatureClass> sampleFeatureClass;
sampleFeatureClass = FdoFeatureClass::Create(L"SampleFeatureClass",
    L"Sample Feature Class Description");
sampleFeatureClass->SetIsAbstract(false);
// get a pointer to the feature class's property collection
// this pointer is used to add data and other properties to the
class
FdoPtr<FdoPropertyDefinitionCollection> sampleFeatureClassPropert
ies;
sampleFeatureClassProperties = sampleFeatureClass->GetProperties();
// get a pointer to the feature schema's class collection
// this object is used to add classes to the schema
FdoPtr<FdoClassCollection> sampleClassCollection;
sampleClassCollection = sampleFeatureSchema->GetClasses();
// get a pointer to the feature class's identity property collec
tion
// this property is used to add identity properties to the feature
// class
FdoPtr<FdoDataPropertyDefinitionCollection> sampleFeatureClassId
entityProperties;
sampleFeatureClassIdentityProperties = sampleFeatureClass->
GetIdentityProperties();
// create a data property that is of type Int32 and identifies
// the feature uniquely
FdoPtr<FdoDataPropertyDefinition> sampleIdentityDataProperty;
sampleIdentityDataProperty = FdoDataPropertyDefinition::Cre
ate(L"SampleIdentityDataProperty", L"Sample Identity Data Property
Description");
sampleIdentityDataProperty->SetDataType(FdoDataType_Int32);

```

```

sampleIdentityDataProperty->SetReadOnly(false);
sampleIdentityDataProperty->SetNullable(false);
sampleIdentityDataProperty->SetIsAutoGenerated(false);
// add the identity property to the sampleFeatureClass
sampleFeatureClassProperties->Add(sampleIdentityDataProperty);
sampleFeatureClassIdentityProperties->Add(sampleIdentityDataProperty);
// create a data property that is of type String and names the
// feature
FdoPtr<FdoDataPropertyDefinition> sampleNameDataProperty;
sampleNameDataProperty = FdoDataPropertyDefinition::Create(L"SampleNameDataProperty", L"Sample Name Data Property Description");
sampleNameDataProperty->SetDataType(FdoDataType_String);
sampleNameDataProperty->SetLength(64);
sampleNameDataProperty->SetReadOnly(false);
sampleNameDataProperty->SetNullable(false);
sampleNameDataProperty->SetIsAutoGenerated(false);
// add the name property to the sampleFeatureClass
sampleFeatureClassProperties->Add(sampleNameDataProperty);
// create a geometric property
FdoPtr<FdoGeometricPropertyDefinition> sampleGeometricProperty;
sampleGeometricProperty = FdoGeometricPropertyDefinition::Create(L"SampleGeometricProperty", L"Sample Geometric Property Description");
sampleGeometricProperty->SetGeometryTypes(FdoGeometricType_Surface);
sampleGeometricProperty->SetReadOnly(false);
sampleGeometricProperty->SetHasMeasure(false);
sampleGeometricProperty->SetHasElevation(false);
// add the geometric property to the sampleFeatureClass
sampleFeatureClassProperties->Add(sampleGeometricProperty);
// identify it as a geometry property
sampleFeatureClass->SetGeometryProperty(sampleGeometricProperty);
// add the feature class to the schema
sampleClassCollection->Add(sampleFeatureClass);
// point the ApplySchema command at the newly created feature
// schema and execute
sampleApplySchema->SetFeatureSchema(sampleFeatureSchema);
sampleApplySchema->Execute();

```

Example: Describing a Schema and Writing It to an XML File

The following sample code demonstrates describing a schema and writing it to an XML file:

```
// create the DescribeSchema command
FdoPtr<FdoIDescribeSchema> sampleDescribeSchema;
sampleDescribeSchema = (FdoIDescribeSchema *)
    connection->CreateCommand(FdoCommandType_DescribeSchema);
// executing the DescribeSchema command returns a feature
// schema collection that is, the set of feature schema which
// reside in the DataStore
FdoPtr<FdoFeatureSchemaCollection> sampleFeatureSchemaCollection;
sampleFeatureSchemaCollection = sampleDescribeSchema->Execute();
// find the target feature schema in the collection, write it
// to an xml file, and clear the collection
sampleFeatureSchema = sampleFeatureSchemaCollection->Find
Item(L"SampleFeatureSchema");
sampleFeatureSchema->WriteXml(L"SampleFeatureSchema.xml");
sampleFeatureSchemaCollection->Clear();
```

Example: Destroying a Schema

The following sample code demonstrates destroying a schema:

```
// create the DestroySchema command
FdoPtr<FdoIDestroySchema> sampleDestroySchema;
sampleDestroySchema = (FdoIDestroySchema *)
    connection->CreateCommand(FdoCommandType_DestroySchema);
// destroy the schema
sampleDestroySchema->SetSchemaName(L"SampleFeatureSchema");
sampleDestroySchema->Execute();
```

Example: Creating a Schema Read In from an XML File

The following sample code demonstrates creating a schema read in from an XML file:

```
sampleFeatureSchemaCollection->ReadXml(L"SampleFeatureSchema.xml");
sampleFeatureSchema = sampleFeatureSchemaCollection->Find
Item(L"SampleFeatureSchema");
sampleApplySchema->SetFeatureSchema(sampleFeatureSchema);
sampleApplySchema->Execute();
sampleFeatureSchemaCollection->Clear();
```

SampleFeatureSchema.xml

The following sample XML schema is the contents of the file written out by the WriteXml method belonging to the FdoFeatureSchema class object that was created in the preceding sample code:

```

<?xml version="1.0" encoding="UTF-8" ?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://fdo_customer/SampleFeatureSchema"
  xmlns:fdo="http://fdo.osgeo.org/schema"
  xmlns:gml="http://www.opengis.net/gml"
  xmlns:SampleFeatureSchema="http://fdo_customer/
    SampleFeatureSchema"
  elementFormDefault="qualified"
  attributeFormDefault="unqualified">
<xs:annotation>
  <xs:documentation>Sample Feature Schema Description
  </xs:documentation>
  <xs:appinfo source="http://fdo.osgeo.org/schema" />
</xs:annotation>
<xs:element name="SampleFeatureClass"
  type="SampleFeatureSchema:SampleFeatureClassType"
  abstract="false" substitutionGroup="gml:_Feature">
  <xs:key name="SampleFeatureClassKey">
    <xs:selector xpath="."/SampleFeatureClass" />
    <xs:field xpath="SampleIdentityDataProperty" />
  </xs:key>
</xs:element>
<xs:complexType name="SampleFeatureClassType"
  abstract="false"
  fdo:geometryName="SampleGeometricProperty"
  fdo:hasMeasure="false"
  fdo:hasElevation="false"
  fdo:srsName="SC_0"
  fdo:geometricTypes="surface">
  <xs:annotation>
    <xs:documentation>Sample Feature Class Description
    </xs:documentation>
    <xs:appinfo source="http://fdo.osgeo.org/schema" />
    <xs:documentation>Sample Geometric Property Descrip
tion</xs:documentation>
  </xs:annotation>
  <xs:complexContent>
    <xs:extension base="gml:AbstractFeatureType">
      <xs:sequence>
        <xs:element name="SampleIdentityDataProperty"
          default=""
          type="fdo:int32">
          <xs:annotation>

```

```

        <xs:documentation>
            Sample Identity Data Property Description
        </xs:documentation>
    </xs:annotation>
</xs:element>
<xs:element name="SampleNameDataProperty"
    default="">
    <xs:annotation>
        <xs:documentation>
            Sample Name Data Property Description
        </xs:documentation>
    </xs:annotation>
    <xs:simpleType>
        <xs:restriction base="xs:string">
            <xs:maxLength value="64" />
        </xs:restriction>
    </xs:simpleType>
</xs:element>
</xs:sequence>
</xs:extension>
</xs:complexContent>
</xs:complexType>
</xs:schema>

```

Data Maintenance

8

This chapter provides information about using the FDO API to maintain data.

Data Maintenance Operations

The primary operations associated with data maintenance are:

- Inserting
- Updating
- Deleting
- Transactions
- Locking

NOTE Discussion of Transactions and Locking is deferred to a future release of this document.

Inserting Values

Preconditions

In a previous chapter, we created a feature schema and added a feature class to it. The feature class had three properties: an integer data property, a string data property, and a geometric property. We applied this feature schema to the data store. We are now ready to create feature data objects, which are instances of the feature class, and insert them into the data store.

Property Values in General

We can now create feature data objects, which are instances of the feature class, by defining a set of property values corresponding to the properties defined for the class and then inserting them into the data store.

An FDO class corresponds roughly to a table definition in a relational database and a property of a class corresponds roughly to a column definition in a table. Adding the property values corresponds roughly to adding a row in the table.

The main distinction between a data value or geometry value and a property value is the order in which they are created. A data value or geometry value object is created first and is then used to create a property value object. The property value object is then added to the value collection object belonging to the Insert command object. Then, the command is executed.

An insert operation consists of the following steps:

- 1 Create the insert command object (type `FdoInsert`); this object can be reused for multiple insert operations.
- 2 Point the insert command object at the feature class to which you are adding values (call the `SetFeatureClassName(<className>)` method).
- 3 From the insert command object, obtain a pointer using the `GetPropertyValues()` method to a value collection object (type `FdoPropertyValueCollection`). You will add property values to the insert command object by adding values to the collection object.
- 4 Create a data value (type `FdoDataValue`) or geometry value (type `FdoGeometryValue`) object. Creating the data value is straightforward; you pass the string or integer value to a static `Create()` method. Creating the geometry value is described in [Geometry Property Values](#) on page 81.
- 5 Create a property value (type `FdoPropertyValue`) object, which involves passing the data value or geometry value object as an argument to a static `Create()` method.
- 6 Add the property value object to the value collection object.
- 7 Execute the Insert command.

Data Property Values

A data value object contains data whose type is one of the following:

- Boolean

- Byte
- DateTime
- Decimal
- Double
- Int16
- Int32
- Int64
- Single (another floating point type)
- String
- Binary large object (BLOB)
- Character large object (CLOB)

The data value object is added to the data property value object. The data property value object is added to the property value collection belonging to the Insert command.

Geometry Property Values

A geometry property value object contains a geometry in the form of a byte array. A geometry can be relatively simple, for example, a point (a single pair of ordinates), or quite complex, for example, a polygon (one or more arrays of ordinates). In the latter case, a number of geometry objects are created and then combined together to form the target geometry. Finally, the target geometry is converted to a byte array and incorporated into the geometry property value object.

Creating a geometry value object consists of the following steps:

- 1 Create a geometry value object (type `FdoGeometryValue`) using a static `Create()` method.
- 2 Create a geometry factory object (type `FdoAgfGeometryFactory`) using a static `GetInstance()` method. This object is used to create the geometry object or objects which comprise the target geometry.
- 3 Create the required geometry object or objects using the appropriate `Create<geometry> method()` belonging to the geometry factory object.

- 4 Use the geometry factory object to convert the target geometry object to a byte array.
- 5 Incorporate the byte array into the geometry property value object.

Example: Inserting an Integer, a String, and a Geometry Value

The following sample code shows how to insert an integer, a string, and a geometry value:

```

// create the insert command
FdoPtr<FdoIInsert> sampleInsert;
sampleInsert = (FdoIInsert *)
    connection->CreateCommand(FdoCommandType_Insert);
// index returned by the operation which adds a value to the value
// collection
FdoInt32 valueCollectionIndex = 0;
// point the Insert command to the target class
// use a fully qualified class name
// whose format is <schemaName>:<className>
sampleInsert-> SetFeatureClassName(L"SampleFeatureSchema:SampleFea
tureClass");
// get the pointer to the value collection used to add properties
// to the Insert command
FdoPtr<FdoPropertyValueCollection> samplePropertyValues;
samplePropertyValues = sampleInsert->GetPropertyValues();
// create an FdoDataValue for the identity property value
FdoPtr<FdoDataValue> sampleIdentityDataValue;
sampleIdentityDataValue = FdoDataValue::Create(101);
// add the FdoDataValue to the identity property value
FdoPtr<FdoPropertyValue> sampleIdentityPropertyValue;
sampleIdentityPropertyValue =
    FdoPropertyValue::Create(L"SampleIdentityDataProperty",
        sampleIdentityDataValue);
// add the identity property value to the value collection
valueCollectionIndex =
    samplePropertyValues->Add(sampleIdentityPropertyValue);
// create an FdoDataValue for the name property value
FdoPtr<FdoDataValue> sampleNameDataValue;
sampleNameDataValue = FdoDataValue::Create(L"Blue Lake");
// add the FdoDataValue to the name property value
FdoPtr<FdoPropertyValue> sampleNamePropertyValue;
sampleNamePropertyValue =
    FdoPropertyValue::Create(L"SampleNameDataProperty",
        sampleNameDataValue);
// add the name property value to the value collection
valueCollectionIndex =
    samplePropertyValues->Add(sampleNamePropertyValue);
// create an FdoGeometryValue for the geometry property value
// this polygon represents a lake which has an island
// the outer shoreline of the lake is defined as a linear ring
// the shoreline of the island is defined as a linear ring
// the outer shoreline is the external boundary of the polygon

```

```

// the island shoreline is an internal linear ring
// a polygon geometry can have zero or more internal rings
FdoPtr<FdoGeometryValue> sampleGeometryValue;
sampleGeometryValue = FdoGeometryValue::Create();
// create an instance of a geometry factory used to create the
// geometry objects
FdoPtr<FdoFgfGeometryFactory> sampleGeometryFactory;
sampleGeometryFactory = FdoFgfGeometryFactory::GetInstance();
// define the external boundary of the polygon, the shoreline of
// Blue Lake
FdoPtr<FdoILinearRing> exteriorRingBlueLake;
FdoInt32 numBlueLakeShorelineOrdinates = 10;
double blueLakeExteriorRingOrdinates[] = {52.0, 18.0, 66.0, 23.0,
    73.0, 9.0, 48.0, 6.0, 52.0, 18.0};
exteriorRingBlueLake = sampleGeometryFactory->CreateLinearRing(
    FdoDimensionality_XY, numBlueLakeShorelineOrdinates,
    blueLakeExteriorRingOrdinates);
// define the shoreline of Goose Island which is on Blue Lake
// this is the sole member of the list of interior rings
FdoPtr<FdoILinearRing> linearRingGooseIsland;
FdoInt32 numGooseIslandShorelineOrdinates = 10;
double gooseIslandLinearRingOrdinates[] = {59.0, 18.0, 67.0, 18.0,
    67.0, 13.0, 59.0, 13.0, 59.0, 18.0};
linearRingGooseIsland = sampleGeometryFactory->CreateLinearRing(
    FdoDimensionality_XY, numGooseIslandShorelineOrdinates,
    gooseIslandLinearRingOrdinates);
// add the Goose Island linear ring to the list of interior rings
FdoPtr<FdoLinearRingCollection> interiorRingsBlueLake;
interiorRingsBlueLake = FdoLinearRingCollection::Create();
interiorRingsBlueLake->Add(linearRingGooseIsland);
// create the Blue Lake polygon
FdoPtr<FdoIPolygon> blueLake;
blueLake =
    sampleGeometryFactory->CreatePolygon(exteriorRingBlueLake,
    interiorRingsBlueLake);
// convert the Blue Lake polygon into a byte array
// and set the geometry value to this byte array
FdoByteArray * geometryByteArray =
    sampleGeometryFactory->GetAgf(blueLake);
sampleGeometryValue->SetGeometry(geometryByteArray);
// add the Blue Lake FdoGeometryValue to the geometry property
value
FdoPtr<FdoPropertyValue> sampleGeometryPropertyValue;

```

```

sampleGeometryPropertyValue =
    FdoPropertyValue::Create (L"SampleGeometryProperty",
        sampleGeometryValue);
// add the geometry property value to the value collection
valueCollectionIndex =
    samplePropertyValues->Add(sampleGeometryPropertyValue);
// do the insertion
// the command returns an FdoIFeatureReader
FdoPtr<FdoIFeatureReader sampleFeatureReader;
sampleFeatureReader = sampleInsert->Execute();

```

Updating Values

After inserting (see [Inserting Values](#) on page 79), you can update the values. The update operation involves identifying a feature class (“table”), a feature class object (“row”), and an object property (“column in a row”) to be changed, and supplying a new value for the object property to replace the old.

First, create an `FdoUpdate` command object and use the command object’s `SetFeatureClassName()` method to identify the feature class. Then, create a filter to identify the feature class object whose properties we want to update, and use the command object’s `SetFilter()` method to attach the command to it. Filters are discussed in [Filter and Expression Languages](#) on page 99.

One of the data properties in the example `SampleFeatureClass` class definition is an identity property, whose name is “`SampleIdentityDataProperty`” and whose type is `fdo:Int32`. This means that its value uniquely identifies the feature class object, that is, the “row”. Use the name of the identity property in the filter. In the Insert operation, the value of the identity property was set to be ‘101’. The value of the filter that is needed is “(`SampleIdentityDataProperty = 101`)”.

Finally, create a property value, which contains the new value, attach it to the command object, and then execute the command.

Example: Updating Property Values

The following is an example of updating property values:

```

FdoPtr<FdoIUpdate> sampleUpdate;
sampleUpdate =
    (FdoIUpdate *)connection->CreateCommand(FdoCommandType_Update);
FdoInt32 numUpdated = 0;
// point the Update command at the target feature class
// use a fully qualified class name
// whose format is <schemaName>:<className>
sampleUpdate->SetFeatureClassName(L"SampleFeatureSchema:SampleFeatureClass");
// set the filter to identify which set of properties to update
sampleUpdate->SetFilter(L"( SampleIdentityDataProperty = 101 )");
// get the pointer to the value collection used to add properties
// to the Update command
// we are reusing the samplePropertyValues object that we used
// for the insert operation
samplePropertyValues = sampleUpdate->GetPropertyValues();
// create an FdoDataValue for the name property value
FdoPtr<FdoDataValue> sampleNameDataValue;
sampleNameDataValue = FdoDataValue::Create(L"Green Lake");
// set the name and value of the property value
sampleNamePropertyValue->SetName(L"SampleNameDataProperty");
sampleNamePropertyValue->SetValue(sampleNameDataValue);
// add the name property value to the property value collection
// owned by the Update command
samplePropertyValues->Add(sampleNamePropertyValue);
// execute the command
numUpdated = sampleUpdate->Execute();

```

Deleting Values

In addition to inserting (see [Inserting Values](#) on page 79) and updating (see [Updating Values](#) on page 85) values, you can delete the values. The deletion operation involves identifying a feature class (“table”) whose feature class objects (“rows”) are to be deleted.

First, create an `FdoIDelete` command object and use the command object’s `SetFeatureClassName()` method to identify the feature class. Then, create a filter to identify the feature class objects that you want to delete, and use the command object’s `SetFilter()` method to attach the filter to it. You can use the same filter that was specified in the preceding section, [Updating Values](#) on page 85. Finally, execute the command.

Example: Deleting Property Values

```
FdoPtr<FdoIDelete> sampleDelete;  
sampleDelete =  
    (FdoIDelete *)connection->CreateCommand(FdoCommandType_Delete);  
FdoInt32 numDeleted = 0;  
sampleDelete->  
    SetFeatureClassName(L"SampleFeatureSchema:SampleFeatureClass");  
sampleDelete->SetFilter(L"( SampleIdentityDataProperty = 101 )");  
numDeleted = sampleDelete->Execute();
```

Related Class Topics

The following classes are used in the preceding Data Maintenance examples:

- FdoInsert
- FdoPropertyValueCollection
- FdoDataValue
- FdoPropertyValue
- FdoGeometryValue
- FdoFgfGeometryFactory
- FdoLinearRing
- FdoLinearRingCollection
- FdoIPolygon
- FdoByteArray
- FdoIDelete
- FdoIUpdate

For more information, see *FDO API Reference Help*.

Performing Queries

9

This chapter describes how to create and perform queries. In the FDO API, you can use queries to retrieve specific features from a data store.

Creating a Query

You create and perform queries using the `FdoISelect` class, which is a member of the Feature sub-package of the Commands package. Queries are used to retrieve features from the data store, and are executed against one class at a time. The class is specified using the `SetFeatureClassName()` method in `FdoIFeatureCommand`. The `SetFeatureClassName` can be used with feature and non-feature classes.

`FdoISelect` supports the use of filters to limit the scope of features returned by the command. This is done through one of the `SetFilter` methods available in the `FdoIFeatureCommand` class. The filter is similar to the SQL WHERE clause, which specifies the search conditions that are applied to one or more class properties.

Search conditions include spatial and non-spatial conditions. Non-spatial queries create a condition against a data property, such as an integer or string. Basic comparisons (`=`, `<`, `>`, `>=`, `<=`, `!=`), pattern matching (like), and 'In' comparisons can be specified. Spatial queries create a spatial condition against a geometry property. Spatial conditions are enumerated in `FdoSpatialCondition` and `FdoDistanceCondition`.

The feature reader (`FdoIFeatureReader`) is used to retrieve the results of a query for feature and non-feature classes. To retrieve the features from the reader, iterate through the reader using the `FdoIFeatureReader.ReadNext` method().

Query Example

In the Data Maintenance chapter, we created an instance of the `FdoFeatureClass SampleFeatureClass` and assigned values to its integer, string, and geometry properties (see [Example: Inserting an Integer, a String, and a Geometry Value](#) on page 82). The sample code in the following query example selects this instance and retrieves the values of its properties. Specifically, the sample code does the following:

- 1 Creates the select command, and
- 2 Points the select command at the target `FdoFeatureClass SampleFeatureClass`, and
- 3 Creates a filter to identify which instance of `SampleFeatureClass` to select, and
- 4 Points the select command at the filter, and
- 5 Executes the command, which returns an `FdoIFeatureReader` object, and
- 6 Loops through the feature reader object, which contains one or more query results depending on the filter arguments. In the sample code provided, there is only one result.
- 7 Finally, the code extracts the property values from each query result.

```

// we have one FdoFeatureClass object in the DataStore
// create a query that returns this object
// create the select command
FdoPtr<FdoISelect> sampleSelect;
sampleSelect = (FdoISelect *)
    connection->CreateCommand(FdoCommandType_Select);
// point the select command at the target FdoFeatureClass
// SampleFeatureClass
sampleSelect->SetFeatureClassName(L"SampleFeatureClass");
// create the filter by
// 1. creating an FdoIdentifier object containing the name of
//    the identity property
FdoPtr<FdoIdentifier> queryPropertyName;
queryPropertyName =
    FdoIdentifier::Create(L"SampleIdentityDataProperty");
// 2. creating an FdoDataValue object containing the value of the
//    identity property
FdoPtr<FdoDataValue> queryPropertyValue;
queryPropertyValue = FdoDataValue::Create(101);
// 3. calling FdoComparisonCondition::Create() passing in the
//    the queryPropertyName, an enumeration constant signifying an
//    equals comparison operation, and the queryPropertyValue
FdoPtr<FdoFilter> filter;
filter = FdoComparisonCondition::Create(queryPropertyName,
    FdoComparisonOperations_EqualTo, queryPropertyValue);
// point the select command at the filter
sampleSelect->SetFilter(filter);
// execute the select command
FdoPtr<FdoIFeatureReader> queryResults;
queryResults = sampleSelect->Execute();
// declare variables needed to capture query results
FdoPtr<FdoClassDefinition> classDef;
FdoPtr<FdoPropertyDefinitionCollection> properties;
FdoInt32 numProperties = 0;
FdoPropertyDefinition * propertyDef;
FdoPropertyType propertyType;
FdoDataType dataType;
FdoDataPropertyDefinition * dataPropertyDef;
FdoString * propertyName = NULL;
FdoPtr<FdoByteArray> byteArray;
FdoIGeometry * geometry = NULL;
FdoGeometryType geometryType = FdoGeometryType_None;
FdoIPolygon * polygon = NULL;

```

```

FdoILinearRing * exteriorRing = NULL;
FdoILinearRing * interiorRing = NULL;
FdoIDirectPosition * position = NULL;
FdoInt32 dimensionality = FdoDimensionality_XY;
FdoInt32 numPositions = 0;
FdoInt32 numInteriorRings = 0;
// loop through the query results
while (queryResults->ReadNext()) {
    // get the feature class object and its properties
    classDef = queryResults->GetClassDefinition();
    properties = classDef->GetProperties();
    // loop through the properties
    numProperties = properties->GetCount();
    for(int i = 0; i < numProperties; i++) {
        propertyDef = properties->GetItem(i);
        // get the property name and property type
        propertyName = propertyDef->GetName();
        propertyType = propertyDef->GetPropertyType();
        switch (propertyType) {
            // it's a data property
            case FdoPropertyType_DataProperty:
                dataPropertyDef =
                    dynamic_cast<FdoDataPropertyDefinition *>
                        (propertyDef);
                dataType = dataPropertyDef->GetDataType();
                switch (dataType) {
                    case FdoDataType_Boolean:
                        break;
                    case FdoDataType_Int32:
                        break;
                    case FdoDataType_String:
                        break;
                    default:
                }
                break;
            // it's a geometric property
            // convert the byte array to a geometry
            // and determine the derived type of the geometry
            case FdoPropertyType_GeometricProperty:
                byteArray = queryResults->GetGeometry(propertyName);
                geometry =
                    sampleGeometryFactory->CreateGeometryFromAgf
                        (byteArray);

```

```

geometryType = geometry->GetDerivedType();
// resolve the derived type into a list of ordinates
switch (geometryType) {
    case FdoGeometryType_None:
        break;
    case FdoGeometryType_Point:
        break;
    case FdoGeometryType_LineString:
        break;
    case FdoGeometryType_Polygon:
        polygon = dynamic_cast<FdoIPolygon *>(geometry);
        exteriorRing = polygon->GetExteriorRing();
        dimensionality = exteriorRing-
            >GetDimensionality();
        numPositions = exteriorRing->GetCount();
        double X, Y, Z, M;
        for(int i=0; i<numPositions; i++) {
            position = exteriorRing->GetItem(i);
            if (dimensionality & FdoDimensionality_Z &&
                dimensionality & FdoDimensionality_M) {
                X = position->GetX();
                Y = position->GetY();
                Z = position->GetZ();
                M = position->GetM();
            }
            else if (dimensionality & FdoDimensionality_Z
                && !(dimensionality & FdoDimensionality_M)) {
                X = position->GetX();
                Y = position->GetY();
                Z = position->GetZ();
            }
            else {
                X = position->GetX();
                Y = position->GetY();
            }
        }
        numInteriorRings = polygon-
            >GetInteriorRingCount();
        for(int i=0; i<numInteriorRings; i++) {
            interiorRing = polygon->GetInteriorRing(i);
            // do same for interior ring as exterior ring
        }
        break;
    case FdoGeometryType_MultiPoint:
        break;
}

```

```

        case FdoGeometryType_MultiLineString:
            break;
        case FdoGeometryType_MultiPolygon:
            break;
        case FdoGeometryType_MultiGeometry:
            break;
        case FdoGeometryType_CurveString:
            break;
        case FdoGeometryType_CurvePolygon:
            break;
        case FdoGeometryType_MultiCurveString:
            break;
        case FdoGeometryType_MultiCurvePolygon:
            break;
        default:
    }
    break;
default:
}
}
}

```

Long Transaction Processing

10

This chapter defines long transactions (LT) and long transaction interfaces, and explains how to implement LT processing in your application.

NOTE For this release, the providers that support long transaction processing are Autodesk FDO Provider for Oracle and OSGeo FDO Provider for ArcSDE.

What Is Long Transaction Processing?

A long transaction (LT) is an administration unit that is used to group conditional changes to objects. Depending on the situation, such a unit can contain conditional changes to one or to many objects. Long transactions are used to modify as-built data in the database without permanently changing the as-built data. Long transactions can be used to apply revisions or alternates to an object.

A root long transaction is a long transaction that represents permanent data and that has descendents. Any long transaction has a root long transaction as an ancestor in its long transaction dependency graph. A leaf long transaction does not have descendents.

For more information about Oracle-specific long transaction versions and locking, see [Locking and Long Transactions](#) on page 136.

Supported Interfaces

In the current release of FDO, the following long transaction interfaces are supported:

- `FDOIActivateLongTransaction`
- `FDOIDeactivateLongTransaction`
- `FDOIRollbackLongTransaction`
- `FDOICommitLongTransaction`
- `FDOICreateLongTransaction`
- `FDOIGetLongTransaction`

These interfaces are summarized below. For more information about their usage, supported methods, associated enumerations and readers, see the *FDO API Reference Help*.

FDOIActivateLongTransaction

The `FdoiActivateLongTransaction` interface defines the `ActivateLongTransaction` command, which activates a long transaction where feature manipulation and locking commands operate on it. Input to the activate long transaction command is the long transaction name. The `Execute` operation activates the identified long transaction.

FDOIDeactivateLongTransaction

The `FdoiDeactivateLongTransaction` interface defines the `DeactivateLongTransaction` command, which deactivates the active long transaction where feature manipulation and locking commands operate on it. If the active long transaction is the root long transaction, then no long transaction will be deactivated.

FDOIRollbackLongTransaction

The `FdoiRollbackLongTransaction` interface defines the `RollbackLongTransaction` command, which allows a user to execute rollback operations on a long transaction. Two different rollback operations are available: Full and Partial.

The operation is executed on all data within a long transaction and on all its descendents. The data is removed from the database and all versions involved in the process deleted.

NOTE If the currently active long transaction is the same as the one being committed or rolled back, then, if the commit or rollback succeeds, the provider resets the current active long transaction to be the root long transaction. If it does not succeed, the active long transaction is left alone and current. If the currently active long transaction is not the same as the one being committed or rolled back, then it is not affected.

FDOICCommitLongTransaction

The FdoICommitLongTransaction interface defines the CommitLongTransaction command, which allows a user to execute commit operations on a long transaction. Two different commit operations are available: Full and Partial.

The commit operation can be performed on a leaf long transaction only. A long transaction is a leaf long transaction if it does not have descendents.

FDOICreateLongTransaction

The FdoICreateLongTransaction interface defines the CreateLongTransaction command, which creates a long transaction that is based on the currently active long transaction. There is always an active long transaction. If the user has not activated a user-defined long transaction, then the root long transaction is active.

Input to the CreateLongTransaction command includes a name and description for the new long transaction. The long transaction name submitted to the command has to be unique. If it is not unique, an exception is thrown.

FDOIGetLongTransactions

The FdoIGetLongTransactions interface defines the GetLongTransactions command, which allows the user to retrieve long transaction information. If a long transaction name is submitted, the command returns the information for the named long transaction only. If no long transaction name is given, the command retrieves the names of all available long transactions.

For each returned long transaction, the user has the option to retrieve a list of descendents and/or ancestors.

Filter and Expression Languages



This chapter discusses the use of filters and filter expressions. You can use filters and expressions to specify to an FDO provider how to identify a subset of the objects in a data store.

For more information and implementation details about the expression functions signatures, the RDBMS-specific built-in support for some of the functions, and the provider-specific support, see the appendix [Expression Functions](#) on page 221.

Filters

FDO uses filters through its commands (including provider-specific commands) to select certain features and exclude others.

A filter is a construct that an application specifies to an FDO provider to identify a subset of objects of an FDO data store. For example, a filter may be used to identify all Road type features that have 2 lanes and that are within 200 metres of a particular location. Many FDO commands use filter parameters to specify the objects to which the command applies. For example, a select command takes a filter to identify the objects that the application wants to retrieve or a delete command takes a filter to identify the objects that the application wants to delete from the data store.

When a command executes, the filter is evaluated for each feature instance and that instance is included in the scope of the command only if the filter evaluates to True. Filters may be specified either as text or as an expression tree. Feature providers declare their level of support for filters through the filter capabilities metadata. Query builders should configure themselves based on the filter capabilities metadata in order to provide users with a robust user interface. For more information, see [What Is an Expression?](#) on page 19.

Expressions

FDO uses expressions through its commands (including provider-specific commands) to specify input values in order to filter features. In general, commands in FDO do not support the SQL command language (the one exception is the optional `SQLCommand`). However, to facilitate ease of use for application developers, expressions in FDO can be specified using a textual notation that is based syntactically on expressions and SQL WHERE clauses. In FDO, expressions are not intended to work against tables and columns, but against feature classes, properties, and relationships. For example, an expression to select roads with four or more lanes might look like this:

```
Lanes >= 4
```

An expression is a construct that an application can use to build up a filter. In other words, an expression is a clause of a filter or larger expression. For example, “`Lanes >=4 and PavementType = 'Asphalt'`” takes two expressions and combines them to create a filter.

Filter and Expression Text

In general, commands in FDO do not support the SQL command language (the one exception is the optional `SQLCommand`). However, to facilitate ease of use for application developers, expressions and filters in FDO can be specified using a textual notation that is based syntactically on expressions and SQL WHERE clauses. The biggest difference between this approach and SQL is that these clauses are not intended to work against tables and columns, but against feature classes, properties, and relationships. For example, a filter to select roads with four or more lanes might look like:

```
Lanes >= 4
```

Similarly, a filter to select all `PipeNetworks` that have at least one `Pipe` in the proposed state might look like:

```
Pipes.state = "proposed"
```

Furthermore, a filter to select all existing parcels whose owner contains the text “Smith” might look like:

```
state = "existing" and owner like "%Smith%"
```

Finally, a filter to select all parcels that are either affected or encroached upon by some change might look like:

```
state in ("affected", "encroached")
```

Language Issues

There are a number of language issues to be considered when working with classes in the Filter, Expression, and Geometry packages:

- Provider-specific constraints on text
- Filter grammar
- Expression grammar
- Filter and Expression keywords
- Data types
- Operators
- Special characters
- Geometry value

Provider-Specific Constraints on Filter and Expression Text

Some providers may have reserved words that require special rules when used with filters and expressions. For more information, see [Oracle Reserved Words Used with Filter and Expression Text](#) on page 135.

Filter Grammar

The rules for entering filter expressions are described in the following sections using BNF notation. For more information about BNF notation, see <http://cui.unige.ch/db-research/Enseignement/analyseinfo/AboutBNF.html>.

The `FdoFilter::Parse()` method supports the following filter grammar:

```

<Filter> ::= '(' Filter ')'
| <LogicalOperator>
| <SearchCondition>
<LogicalOperator> ::= <BinaryLogicalOperator>
| <UnaryLogicalOperator>
<BinaryLogicalOperator> ::=
<Filter> <BinaryLogicalOperations> <Filter>
<SearchCondition> ::= <InCondition>
| <ComparisonCondition>
| <GeometricCondition>
| <NullCondition>
<InCondition> ::= <Identifier> IN '(' ValueExpressionCollection
')'
<ValueExpressionCollection> ::= <ValueExpression>
| <ValueExpressionCollection> ',' <ValueExpression>
<ComparisonCondition> ::=
<Expression> <ComparisonOperations> <Expression>
<GeometricCondition> ::= <DistanceCondition>
| <SpatialCondition>
<DistanceCondition> ::=
<Identifier> <DistanceOperations> <Expression> <distance>
<NullCondition> ::= <Identifier> NULL
<SpatialCondition> ::=
<Identifier> <SpatialOperations> <Expression>
<UnaryLogicalOperator> ::= NOT <Filter>
<BinaryLogicalOperations> ::= AND | OR
<ComparisonOperations> ::=
= // EqualTo (EQ)
<> // NotEqualTo (NE)
> // GreaterThan (GT)
>= // GreaterThanOrEqualTo (GE)
< // LessThan (LT)
<= // LessThanOrEqualTo (LE)
LIKE // Like
<DistanceOperations> ::= BEYOND | WITHINDISTANCE
<distance> ::= DOUBLE | INTEGER
<SpatialOperations> ::= CONTAINS | CROSSES | DISJOINT
| EQUALS | INTERSECTS | OVERLAPS | TOUCHES | WITHIN | COVEREDBY |
INSIDE

```

Expression Grammar

The `FdoExpression::Parse()` method supports the following expression grammar:

```
<Expression> ::= '(' Expression ')'  
| <UnaryExpression>  
| <BinaryExpression>  
| <Function>  
| <Identifier>  
| <ValueExpression>  
<BinaryExpression> ::=  
<Expression> '+' <Expression>  
| <Expression> '-' <Expression>  
| <Expression> '*' <Expression>  
| <Expression> '/' <Expression>  
<DataValue> ::=  
TRUE  
| FALSE  
| DATETIME  
| DOUBLE  
| INTEGER  
| STRING  
| BLOB  
| CLOB  
| NULL  
<Function> ::= <Identifier> '(' <ExpressionCollection> ')'  
<ExpressionCollection> ::=  
| <Expression>  
| <ExpressionCollection> ',' <Expression>  
<GeometryValue> ::= GEOMFROMTEXT '(' STRING ')'  
<Identifier> ::= IDENTIFIER  
<ValueExpression> ::= <LiteralValue> | <Parameter>;  
<LiteralValue> ::= <GeometryValue> | <DataValue>  
<Parameter> ::= PARAMETER | ':' STRING  
<UnaryExpression> ::= '-' <Expression>
```

Expression Operator Precedence

The precedence is shown in YACC notation, that is, the highest precedence operators are at the bottom.

```
%left Add Subtract  
%left Multiply Divide  
%left Negate
```

Filter and Expression Keywords

The following case-insensitive keywords are reserved in the language, that is, they cannot be used as identifier or function names:

```
AND BEYOND COMPARE CONTAINS COVEREDBY CROSSES DATE  
DISJOINT DISTANCE EQUALS FALSE GeomFromText IN INSIDE  
INTERSECTS LIKE NOT NULL OR OVERLAPS RELATE SPATIAL TIME  
TIMESTAMP TOUCHES TRUE WITHIN WITHINDISTANCE
```

Data Types

The available data types are described in this section.

Identifier

An identifier can be any alphanumeric sequence of characters other than a keyword. Identifiers can be enclosed in double quotes to allow special characters and white space. If you need to include a double quote character inside an identifier, double the character, for example "abc""def".

Parameter

Parameters are defined by a colon followed by alphanumeric characters. The FDO filter language extends SQL to allow for a literal string to follow the colon to allow blanks (and other possibilities), for example, :'Enter Name'.

Determine whether parameters are supported by the FDO Provider you are using by checking SupportParameters on the Connection interface.

String

Strings are literal constants enclosed in single quotes. The FDO filter language also supports the special characters (left and right single quotes) that Microsoft Word uses to automatically replace the single quote character typed from the keyboard. If you need to include a single quote character inside a string you can double the character, for example 'aaa"bbb'.

Integer

Integers allow only decimal characters with an optional unary minus sign. Unary plus is not supported.

(-)[0-9]

Double

Floating point numbers have a decimal point, can be signed (-), and include an optional exponent (e[0-9]).

NOTE If an integer is out of the 32-bit precision range, it is converted to floating point.

Examples:

```
-3.4
12345678901234567
1.2e13
```

DateTime

Date and time are parsed using the standard SQL literal strings:

```
DATE 'YYYY-MM-DD'
TIME 'HH:MM:SS[.sss]'
TIMESTAMP 'YYYY-MM-DD HH:MM:SS[.sss]'
```

For example:

```
DATE '1971-12-24'
TIMESTAMP '2003-10-23 11:00:02'
```

NOTE The BLOB and CLOB strings are currently not supported. If you need to support binary input, use parameters.

Operators

The following operators are special characters common to SQL and most programming languages:

Binary Operations

These binary operations are available:

+ Add (for compatibility with SQL string concatenation may also be defined using "||")

- Subtract

* Multiply

/ Divide

Unary Operations

These unary operation are available:

- Negate

Comparison Operations

These comparison operations are available:

= EqualTo (EQ)

<> NotEqualTo (NE)

> GreaterThan (GT)

>= GreaterThanOrEqualTo (GE)

< LessThan (LT)

<= LessThanOrEqualTo (LE)

Operator Precedence

The following precedence is shown from highest to lowest:

Negate NOT

Multiply Divide

Add Subtract

EQ NE GT GE LT LE

AND

OR

Special Character

The following special characters are used in ExpressionCollections and ValueExpressions to define function arguments and IN conditions:

(Left Parenthesis

, Comma

) Right Parenthesis

The Colon (:) is used in defining parameters and the Dot (.) can be included in real numbers and identifiers.

Geometry Value

Geometry values are handled using a function call `GeomFromText('FGF Text string')`, as is typical in an SQL query.

The Autodesk extension to WKT, referred to as FGF Text, is a superset of WKT (that is, you can enter WKT as valid FGF Text strings). Dimensionality is optional. It can be XY, XYM, XYZ, or XYZM. If it is not specified, it is assumed to be XY. For more information about FGF Text, see [FGF Text](#) on page 118.

NOTE Extra ordinates are ignored, rather than generating an error during FGF text parsing. For example, in the string “POINT (10 11 12)”, the ‘12’ is ignored because the dimensionality is assumed to be XY.

The following is the grammar definition for FGF Text:

<FGF Text> ::= POINT <Dimensionality> <PointEntity>

| LINESTRING <Dimensionality> <LineString>

| POLYGON <Dimensionality> <Polygon>

| CURVESTRING <Dimensionality> <CurveString>

| CURVEPOLYGON <Dimensionality> <CurvePolygon>

| MULTIPOINT <Dimensionality> <MultiPoint>

```

| MULTILINESTRING <Dimensionality> <MultiLineString>
| MULTIPOLYGON <Dimensionality> <MultiPolygon>
| MULTICURVESTRING <Dimensionality> <MultiCurveString>
| MULTICURVEPOLYGON <Dimensionality> <MultiCurvePolygon>
| GEOMETRYCOLLECTION <GeometryCollection>
<PointEntity> ::= '(' <Point> ')'
<LineString> ::= '(' <PointCollection> ')'
<Polygon> ::= '(' <LineStringCollection> ')'
<MultiPoint> ::= '(' <PointCollection> ')'
<MultiLineString> ::= '(' <LineStringCollection> ')'
<MultiPolygon> ::= '(' <PolygonCollection> ')'
<GeometryCollection : '(' <FGF Collection Text> ')'
<CurveString> ::= '(' <Point> '(' <CurveSegmentCollection> ')' ')'
<CurvePolygon> ::= '(' <CurveStringCollection> ')'
<MultiCurveString> ::= '(' <CurveStringCollection> ')'
<MultiCurvePolygon> ::= '(' <CurvePolygonCollection> ')'
<Dimensionality> ::= // default to XY
| XY
| XYZ
| XYM
| XYZM
<Point> ::= DOUBLE DOUBLE
| DOUBLE DOUBLE DOUBLE
| DOUBLE DOUBLE DOUBLE DOUBLE
<PointCollection> ::= <Point>
| <PointCollection ',' <Point>
<LineStringCollection> ::= <LineString>
| <LineStringCollection ',' <LineString>
<PolygonCollection> ::= <Polygon>

```

```

| <PolygonCollection> ',' <Polygon>
<FGF Collection Text> ::= <FGF Text>
| <FGF Collection Text> ',' <FGF Text>
<CurveSegment> ::= CIRCULARARCSEGMENT '(' <Point> ',' <Point> ')'
| LINESTRINGSEGMENT '(' <PointCollection> ')'
<CurveSegmentCollection> ::= <CurveSegment>
| <CurveSegmentCollection> ',' <CurveSegment>
<CurveStringCollection> ::= <CurveString>
| <CurveStringCollection> ',' <CurveString>
<CurvePolygonCollection> ::= <CurvePolygon>
| <CurvePolygonCollection> ',' <CurvePolygon>

```

The only other token type is DOUBLE, representing a double precision floating point values. Integer (non-decimal point) input is converted to DOUBLE in the lexical analyzer.

Examples of the Autodesk extensions include:

```
POINT XY (10 11) // equivalent to POINT (10 11)
```

```
POINT XYZ (10 11 12)
```

```
POINT XYM (10 11 1.2)
```

```
POINT XYZM (10 11 12 1.2)
```

```
GEOMETRYCOLLECTION (POINT xyz (10 11 12),POINT XYM (30 20 1.8),
LINESTRING XYZM(1 2 3 4, 3 5 15, 3 20 20))
```

```
CURVESTRING (0 0 (LINESTRINGSEGMENT (10 10, 20 20, 30 40))))
```

```
CURVESTRING (0 0 (CIRCULARARCSEGMENT (11 11, 12 12),
LINESTRINGSEGMENT (10 10, 20 20, 30 40))))
```

```
CURVESTRING (0 0 (ARC (11 11, 12 12), LINESTRINGSEGMENT (10 10, 20
20, 30 40))))
```

```
CURVESTRING XYZ (0 0 0 (LINESTRINGSEGMENT (10 10 1, 20 20 1, 30 40
1))))
```

```
MULTICURVESTRING ((0 0 (LINESTRINGSEGMENT (10 10, 20 20, 30 40))), (0
0 (ARC (11 11, 12 12), LINESTRINGSEGMENT (10 10, 20 20, 30 40))))
```

```
CURVEPOLYGON ((0 0 (LINESTRINGSEGMENT (10 10, 10 20, 20 20), ARC  
(20 15, 10 10))), (0 0 (ARC (11 11, 12 12), LINESTRINGSEGMENT (10 10, 20  
20, 40 40, 90 90))))
```

```
MULTICURVEPOLYGON (((0 0 (LINESTRINGSEGMENT (10 10, 10 20, 20 20),  
ARC (20 15, 10 10))), (0 0 (ARC (11 11, 12 12), LINESTRINGSEGMENT (10 10,  
20 20, 40 40, 90 90)))), ((0 0 (LINESTRINGSEGMENT (10 10, 10 20, 20 20),  
ARC (20 15, 10 10))), (0 0 (ARC (11 11, 12 12), LINESTRINGSEGMENT (10 10,  
20 20, 40 40, 90 90)))))
```

The Geometry API

12

This chapter describes the FdoGeometry API (hereafter called the “Geometry API”) and explains the various geometry types and formats.

Introduction

The Geometry API supports specific Autodesk applications and APIs, including FDO (Feature Data Objects). This API consists of the following components:

- a Geometry Type package (all through fully encapsulated interfaces)
- an Abstract Geometry Factory
- a Concrete Geometry Factory for FGF

You can work with the Geometry API in several different ways:

- FGF (Feature Geometry Format)
- FGF Text
- Abstract Geometry Factory

FGF and WKB

WKB is a memory layout used to store geometry features. This format was created by the OpenGIS organization to allow the efficient exchange of geometry data between different components in an FDO system. Most pieces of the original specification defining the WKB format are in the document, *99-050.pdf*, the OpenGIS Simple feature specification for OLE/COM that can be found at www.opengis.org.

FGF is an extended version of the Well Known Binary (WKB) format. The two formats differ in the following ways:

- WKB defines a byte order of the data in every piece of geometry. This is stored as a byte field, which may change the memory alignment from word to byte. In FGF, only one memory alignment type is supported, which is the same alignment type used by the .NET framework and Windows; the encoding uses the little-endian byte order format. As a result, the byte flag does not need to be stored.
- WKB is defined as a 2D format only. This is insufficient to represent 3D points, polylines and polygons. In FGF, the dimension flag has been added. In particular, a flag is included for each geometry piece to indicate whether the geometry is 2D, 3D, or even 4D (storing a measure value as used by dynamic segmentation).
- FGF includes geometry types that are not yet covered by any WKB specification.

FGF Binary Specification

In this section, the memory layout of each simple geometry type is described. The format is based on the OGC specification, which is built on the memory layout of a C++ struct. All arrays have a computable size and are inline; they do not point to a different location in memory. This format allows streaming of geometry data.

First, the different data types, their size, and memory layout are discussed.


```

// double == 8byte IEEE double number in little endian encoding.
// int == 4 byte integer in little endian encoding
// the type of the geometry
enum GeometryType : int
{
    None = 0,
    Point = 1,
    LineString = 2,
    Polygon = 3,
    MultiPoint = 4,
    MultiLineString = 5,
    MultiPolygon = 6,
    MultiGeometry = 7,
    CurveString = 10,
    CurvePolygon = 11,
    MultiCurveString = 12,
    MultiCurvePolygon = 13
}

```

Coordinate Types

This is a bit field, for example, `xym == coordinateDimensionality.XY | CoordinateDimensionality.M`. The following sequence defines the type of coordinates used for this object:

```

enum CoordinateDimensionality : FdoInt32
{
    XY = 0,
    Z = 1,
    M = 2
}

```

Basic Geometry

The following sequence establishes the basic pure geometry:

```

struct Geometry
{
    int geomType;
    CoordinateDimensionality type;
}

```

Notation Definition

The following sequence defines a notation used to specify geometries within a byte stream.

```

// Define a notation within this specification
// int PositionSize(geometry)
// {
//   if (geometry.type == CoordinateDimensionality.XY |
//       // CoordinateDimensionality.M ||
//       geometry.type == CoordinateDimensionality.XY |
//       // CoordinateDimensionality.Z)
//     return 3;
//   if (geometry.type == CoordinateDimensionality.XY |
//       // CoordinateDimensionality.M | CoordinateDimensionality.Z)
//     return 4
//   return 2;
// }
struct Point // : Geometry
{
  int geomType; // == GeometryType.Point;
  CoordinateDimensionality type; // all types allowed
  double[] coords; // size = PositionSize(this)
}
struct LineString
{
  int geomType;
  CoordinateDimensionality type;
  int numPts; // >0
  double[] coords; // size = numPts* PositionSize(this)
}
struct MultiPoint
{
  int geomType;
  int numPoints; // > 0
  Point[] points; // size = numPoints
}
struct MultiLineString
{
  int geomType;
  int numLineStrings; // >= 0
  LineString[] lineStrings; // size = numLineStrings
}
// building block for polygons, not geometry by itself
struct LinearRing
{
  int numPts; // >0
  double[] coords; // size = numPts* PositionSize(polygon)
}

```

```

}
struct Polygon
{
    int geomType;
    CoordinateDimensionality type;
    int numRings; // >= 1 as there has to be at least one ring
    LinearRing[] lineStrings; // size = numRings
}
struct MultiPolygon
{
    int geomType;
    int numPolygons; // >= 0
    Polygon[] polygons; // size = numPolygons
}
struct MultiGeometry
{
    int geomType;
    int numGeom; // >= 0
    Geometry[] geometry; // size = numGeom
}
enum CurveElementType : int
{
    LineString = 1,
    CircularArc = 2
}
struct CurveStringElement
{
    int CurveElementType;
}
struct LinearCurveStringElement
{
    int CurveElementType;
    int length;
    double[] coords; // size = this.length * PositionSize (this)
}
struct CircularArcCurveStringElement
{
    int CurveElementType; // == CurveElmentType.Arc
    double[] coords; // size = 2 * PositionSize(this)
}
struct CurveString
{
    int geomType;

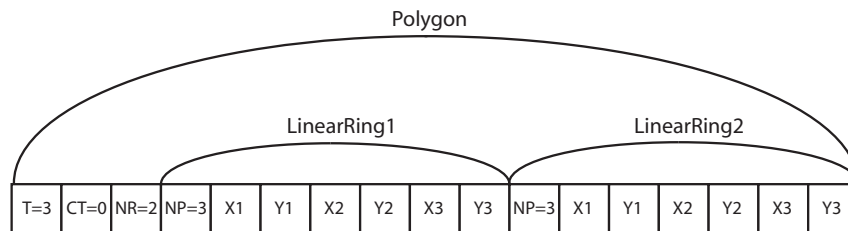
```

```

CoordinateDimensionality type; // all types allowed
double[] startPoint; // size = PositionSize(this)
int numElements; // >=0
CurveStringElement[] elements; // size = numElements
}
struct Ring
{
double[] startPoint; // size = PositionSize(this)
int numElements; // >=0
CurveStringElement[] elements; // size = numElements
}
struct MultiCurveString
{
int geomType;
int numCurveStrings; // >= 0
CurveString[] curveStrings; // size = numCurveStrings
}
struct CurvePolygon
{
int geomType; ;
CoordinateDimensionality type;
int numRings; // >=1 as there has to be at least one ring
Ring[] rings; // size = numRings
}
struct MultiCurvePolygon
{
int geomType;
int numPolygons; // >=0
CurvePolygon[] polygons; // size = numElements
}

```

In the following example a polygon is formatted within a byte array representing the stream according to the FGF specification.



T = 3 stands for GeometryType == GeometryType.Polygon

CT = 0 stands for CoordinateDimensionality == CoordinateDimensionality.XY

NR = 2 stands for number of rings = 2

NP = 3 stands for number of points = 3

FGF Text

FGF Text is the textual analogue to the binary FGF format. It is a superset of the OGC WKT format. XY dimensionality is the default, and is optional. FGF Text can be used to represent any geometry value in the Geometry API, whether or not it originates from the FGF geometry factory. Conversions are done with the following methods:

- `FdoGeometryFactoryAbstract:: CreateGeometry(FdoString* text);`
- `FdoGeometry:: GetText();`

A BNF for the FGF textual specification is contained in the topic [Geometry Value](#) on page 107.

Abstract and Concrete Classes

The Geometry API is almost completely abstract. It provides an object-oriented interface to geometry values. All objects in the Geometry API have factory methods in the `FdoGeometryFactoryAbstract` class. One default implementation is provided, based on FGF in-memory binary storage. It is accessible via the concrete class `FdoFgfGeometryFactory`.

NOTE The `FdoFgfGeometryFactory` employs object pooling for many of the data types in the API. While many methods appear to be executing “Create” or “Get” actions, they are, in fact, accessing object pools, thus avoiding costly operations on the memory heap.

All of the other classes in the Geometry API with the exception of two relate to the main abstract type, `FdoGeometry`. They either derive from it or are components of it.

The two exception concrete classes are:

- `FdoDirectPositionImpl`, a small helper class implementing `FdoIDirectPosition`.
- `FdoEnvelopeImpl`, a small helper class implementing `FdoIEnvelope`.

Geometries in FGF format can be exchanged between software components without depending on the Geometry API itself, because they are not genuine geometry “objects.” FGF content is based on byte arrays. It is handled through a simple `FdoByteArray` class that is not specific to geometry.

Geometry Types

The Geometry types comprise the Global Enum `FdoGeometryType`. The following are Geometry types:

- 0 `FdoGeometryType_None` Indicates no specific type; used for “unknown”, “do not care” or an incompletely constructed Geometry object.

NOTE `FdoGeometryType_None` does not represent an instantiable type. An FDO client should not expect an FDO provider to list support for it in its capabilities.

- 1 `FdoGeometryType_Point` Point type (`FdoIPoint`).
- 2 `FdoGeometryType_LineString` LineString type (`FdoILineString`).
- 3 `FdoGeometryType_Polygon` Polygon type (`FdoIPolygon`).
- 4 `FdoGeometryType_MultiPoint` MultiPoint type (`FdoIMultiPoint`).
- 5 `FdoGeometryType_MultiLineString` MultiLineString type (`FdoIMultiLineString`).
- 6 `FdoGeometryType_MultiPolygon` MultiPolygon type (`FdoIMultiPolygon`).
- 7 `FdoGeometryType_MultiGeometry` MultiGeometry type (`FdoIMultiGeometry`).
- 10 `FdoGeometryType_CurveString` CurveString type (`FdoICurveString`).
- 11 `FdoGeometryType_CurvePolygon` CurvePolygon type (`FdoICurvePolygon`).
- 12 `FdoGeometryType_MultiCurveString` MultiCurveString type (`FdoIMultiCurveString`).
- 13 `FdoGeometryType_MultiCurvePolygon` MultiCurvePolygon type (`FdoIMultiCurvePolygon`).

Mapping Between Geometry and Geometric Types

The FDO API `GeometricType` enumeration of `GeometricProperty` gives the client application some knowledge of which geometry types comprise the geometric property so that it can present the user with an intelligent editor for selecting styles for rendering the geometry. In particular, `GeometricType` relates to shape dimensionality of geometries allowed in FDO geometric properties. The nearest analogues in the Geometry API are:

- `FdoDimensionality`, which pertains to ordinate (not shape) dimensionality of geometry values.
- `FdoGeometryType`, which has types whose abstract base types map to `GeometricType`

The `GeometricType` enumeration is as follows:

- `Point = 0x01`, // Point Type Geometry
- `Curve = 0x02`, // Line and Curve Type Geometry
- `Surface = 0x04`, // Surface (or Area) Type Geometry
- `Solid = 0x08`, // Solid Type Geometry

NOTE The enumeration defines a bit mask and the `GetGeometricTypes` and `SetGeometricTypes` methods take and return an integer. This is to allow a geometry property to be of more than one type. For example, the call:

```
geometricProperty.SetGeometricTypes(Point | Surface);
```

would allow the geometric property to represent either point type geometry or surface type geometry (polygons).

Spatial Context

Spatial Context is a coordinate system with an identity. Any geometries that are to be spatially related must be in a common spatial context.

Providing an identify for each coordinate system supports separate workspaces, such as schematic diagrams, which are non-georeferenced. However, there are also georeferenced cases. In general, two users may create drawings using the same default spatial parameters (for example, rectangular and

10,000x10,000) that have nothing to do with each other. If their drawings are to be put into a common database, the spatial context capability of FDO preserves the container aspect of the data along with the spatial parameters.

The FDO Spatial Context Commands are part of the FDO API. They support control over Spatial Contexts in the following ways:

- **Metadata control.** Creates and deletes Spatial Contexts.
- **Active Spatial Context.** A session setting to specify which Spatial Context to use by default while storing/retrieving geometries and performing spatial queries.

There is a default Spatial Context for each database. Its attributes (such as coordinate system) are specified when the database is created. This Spatial Context is the active one in any FDO session until a Spatial Context Command is used to change this state. The default Spatial Context's identifier number is 0 (zero).

Spatial contexts have two tolerance attributes: `XYTolerance` and `ZTolerance`. The tolerances are in distance units that depend on the coordinate system in use. Geodetic coordinate systems typically have "on the ground" linear distance units instead of the angular (that is, degrees, minutes or seconds) units used for positional ordinates. The meter is the most common unit. Most non-geodetic systems are rectilinear and use the same unit for positional ordinates and distances, for example, meters or feet.

Specify Dimensionality When Creating Geometries Using String Specifications

When creating a 3D geometry from string specifications, you must specify the Dimensionality argument `xyz` explicitly, because the default dimensionality is XY, and the geometry factory code will only process the first two ordinates.

The following code successfully creates `pointOne` as a 3D point, whereas `pointTwo` is created as a 2D point.

```

FdoFgfGeometryFactory * geometryFactory = FdoFgfGeometryFact
ory::GetInstance();
FdoIPoint * pointOne;
pointOne = dynamic_cast<FdoIPoint*>(geometryFactory->CreateGeometry(L"GeomFrom
Text('POINT XYZ (1 2 3)')"));
FdoPoint * pointTwo;
pointTwo = dynamic_cast<FdoPoint*>(geometryFactory->CreateGeometry(L"GeomFrom
Text('POINT (1 2 3)')"));

xmlFeatureFlags = FdoXmlFeatureFlags.Create( None, FdoXmlFlags.Er
rorLevel_Normal, True, FdoXmlFeatureFlags.ConflictOption_Add )

```

Inserting Geometry Values

For information about geometry property values, see [Geometry Property Values](#) on page 81.

See [Example: Inserting an Integer, a String, and a Geometry Value](#) on page 82 for a code example that shows how to insert a Geometry value.

Autodesk FDO Provider for Oracle



This appendix discusses FDO API development issues that are related to using FDO Provider for Oracle.

What Is FDO Provider for Oracle?

The Feature Data Objects (FDO) API provides access to data in a data store. A provider is a specific implementation of the FDO API that provides access to data in a particular data store. FDO Provider for Oracle provides FDO with access to an Oracle-based data store.

FDO Provider for Oracle API provides custom commands specifically designed to work with the FDO API. For example, using these commands, you can do the following:

- Gather information about a provider.
- Transmit client services exceptions.
- Get lists of accessible data stores.
- Create connection objects.
- Create and dropping spatial indexes.

FDO Provider for Oracle General Requirements

For Autodesk Map 3D users, a pre-requisite for creating schema and managing long transactions is to include the setting WM_ADMIN_ROLE in the user definition.

If a user definition does not have this setting, use the *FDO User Manager Tool* to delete the user definition and then recreate it to include WM_ADMIN_ROLE.

FDO Provider for Oracle Connection

This information supplements the Establishing a Connection chapter. You connect to a data store directly through FDO Provider for Oracle, and the underlying data source for the data store is an Oracle database.

You can connect to the data store in one step if you already know its name. Otherwise, you must connect in two steps.

The minimum required connection parameters for the initial call to Open() are service, username, and password.

The service parameter is the Oracle Net Service Name of an Oracle instance. An instance could be running on your machine or on some other machine in the network. You can use the Oracle Net Manager to identify which Oracle instances are available to you and what their Net Service Names are. In an Oracle 10g installation on a Microsoft Windows XP machine, Net Manager can be accessed with Start ► Programs ► Oracle ► Configuration and Migration Tools ► Net Manager. The connection information for the Net Service Name is contained in a file named *tnsnames.ora*, which is located in the Network/Admin folder in either the Oracle instance or the client installation directory.

Multiple users can access the data store. However, access is password-protected.

An Oracle data source, when accessed by FDO Provider for Oracle, may contain more than one data store. For the first call to Open(), a data store name is optional. If successful, the first call to Open() results in the data store parameter becoming a required parameter and a list of the names of the data stores in the data source becoming available. You must choose a data store and call Open() again.

If you know the name of the data store, you can provide it for the first call to `Open()` and make the connection in one step.

FDO Provider for Oracle and Foreign Schemas

FDO Provider for Oracle supports the creation of foreign schemas. A foreign schema is capable of mapping a table to Oracle instances. This allows users with a pre-existing application (for example, one created with Autodesk GIS Design Server) to map their application to FDO. As a result, both the FDO capability and conventional capability can be used by the same application.

Foreign Schema Settings

In order to use a foreign schema, certain privileges for FDO are required. To support the foreign schema capability, the following settings are required:

- FDO schema instance
- Foreign schema instance
- Oracle identity property

These settings are required for accessing the foreign schema objects (that is, tables, views, and sequences.).

Settings on the FDO Schema Instance

If the foreign schema is on a different Oracle instance, create a PUBLIC database link. A database link is a schema object that enables accessing of objects on another Oracle instance.

Settings on the Foreign Schema Instance

If the foreign schema is on a different Oracle instance, create an FDO user using the same Username and password as on the instance where the FDO schema exists.

NOTE This FDO user does not need to have been granted the `f_user_role` role.

Grant the select, update, delete, and insert privileges on tables (views, sequences) to the FDO user that is mapped to the FDO schema. Note that if the Foreign Schema tables are enabled for Oracle Workspace Manager, the privileges must be granted for the based table.

Oracle Identity Property

When specifying the main identifier for a feature class with FDO Provider for Oracle, the data type must be Int64. It must also have the following settings: ReadOnly=True and Autogenerated=True. If an identifier is not created with these properties, it will be created by FDO Provider for Oracle. Otherwise, an exception may be raised in certain conditions.

If the foreign schema uses a different data type for identifiers, the user must define the identifier as a fdo int64 type with ReadOnly=True and Autogenerated=True in the XML configuration file. If the identifier in the foreign schema uses an Oracle sequence to generate the unique numbers, the sequencename override must be defined in the override XML file.

Read-Write Privileges

If FDO requires read-write privileges to work with a foreign schema, the owner of the foreign schema must grant these privileges. Also, access to the Oracle metaschema on the tables is required.

NOTE The owner must explicitly grant these privileges. These privileges will vary, according to the schema owner.

Foreign Schema Limitations

This section provides information about known limitations of foreign schemas.

Ensuring Valid Views When Applying a Feature Schema Against a Foreign Schema

The ApplySchema command can create invalid views when the feature schema is applied against a foreign schema. When you apply against a foreign schema,

tables and columns are not automatically created if they do not already exist. A view is created, however, in the connected data store that references these foreign tables and columns. ApplySchema succeeds even if certain tables or columns cannot be obtained for various reasons, such as:

- The database link to the foreign tables is invalid.
- The Oracle instance containing the foreign tables is shut down or unreachable due to network problems.
- The foreign table or column simply does not exist.

When one of these situations occurs, ApplySchema creates invalid (dangling) views because these views reference tables or columns that cannot be reached.

Invalid views can occur regardless of whether schema overrides have been specified because the default schema mappings can also reference unreachable tables or columns. For example, if the feature schema being applied has a Pole class with no table name override, the Pole class is mapped to the POLE table in the foreign schema and a POLE view (referencing the POLE table) is created in the data store that the Oracle provider is currently connected to. If the POLE table does not exist, then the POLE view is dangling, or invalid.

Use one of the following procedures to correct invalid views, depending on whether the correct table name was specified (either through a schema override or the default class to table mapping rule):

Procedure When Table Name Is Correct

If the table name is correct, but it is not reachable for the reasons listed above:

- 1 Create the table or make it reachable by fixing the database link, fixing network problems, or starting the Oracle instance that contains the table.
- 2 Recompile the view that references the table.

The Oracle SQL statement for recompiling the view is:

Alter view <view_name> compile;

If there are a number of views to recompile, an alternative method is to use the following procedure for a wrong table name, but only do Steps 1 and 3.

Procedure When Wrong Table Name Is Specified

If the wrong table name is specified, to ensure valid views:

- 1 Destroy the feature schema. As long as the feature schema maps onto a foreign schema, destroying it does not result in loss of data. No tables or columns are dropped, only the referencing views created by FDO Provider for Oracle are dropped.
- 2 Fix the schema overrides to supply the proper table name. In some cases, you may need to add a schema override. For example, if a class named "Pole" corresponds to the foreign table "telco_pole", then a table name override must be specified for the Pole class, since the class and table names differ.
- 3 Re-apply the feature schema.

If your feature schema contains a mixture of classes mapped to foreign tables and classes mapped to non-foreign tables, then the procedure is slightly more complex, especially if any non-foreign table contains data. In this case, the following steps must be done programmatically throughout the FDO API:

- 1 Describe the feature schema using the DescribeSchema command. Retain this description.
- 2 Remove every class, except the one with the wrong table name, from the feature schema returned. However, do not delete the classes (that is, do not call FdoClassDefinition::Delete()).
- 3 Describe the feature schema again (ensure that you retain the feature schema from the first Describe).
- 4 Delete the class with the wrong table name from the feature schema returned by the second Describe (by calling its FdoClassDefinition::Delete() function).
- 5 Fix the schema overrides to supply the proper table name.
- 6 Ensure that FdoIApplySchema::SetIgnoreStates() is set to false, then Apply the feature schema described from Step 3. This deletes the class to repair.
- 7 Ensure that FdoIApplySchema::SetIgnoreStates() is set to true, then Apply the feature schema described from Step 1, along with the schema overrides. This re-creates the class to repair.

Overrides Capable of Causing Invalid Views

Any schema mapping between a feature schema element and a table or column can create invalid views. This is true for mappings specified through schema overrides or for default mappings. The specific schema mappings that can cause invalid views are as follows:

- Class to Table name
- Class to Geometry Column name
- Data Property to Column name
- Geometric Property to Column name
- Object Property to Table name

Table Name Restrictions When Working with a Foreign Schema

The ApplySchema command does not automatically create tables in foreign schemas. Therefore, the table specified for each class must already exist in the foreign schema. The Schema Overrides, specified through ApplySchema.SetPhysicalMapping, must contain a class to table mapping for each class whose table is named differently from the class. No mapping is required for classes where the table name and class name are the same.

Schema Access on a Different Oracle Instance

The following are Oracle limitations on foreign schema access if the schema is on a different (remote) Oracle instance:

- LOB type columns are not supported.
- Versioning and locking using Oracle Workspace Manager are not supported.

FDO Provider for Oracle and Schema Overrides

Schema overrides are supported through the Overrides API that is specific to FDO Provider for Oracle. This API is published as part of the FDO SDK.

Schema Override Set

A schema override set is the set of schema overrides for a particular Feature Schema and FDO Provider.

The top level of a schema override set is very similar to the Feature Schema, itself. There is a root class (`OraclePhysicalSchemaMapping`), with a list of classes and a list of relations. These lists are subsets of the lists in the corresponding Feature Schema. It is not necessary to list every class and relation; list only the ones for which overrides are being specified. `OraclePhysicalSchemaMapping` provides the Oracle-specific implementation of `FdoPhysicalSchemaMapping`.

The methods for these MetaClasses are stripped down from the methods on the corresponding Feature Schema MetaClasses. In the Schema Override set, only name and physical properties are specified. For example, the names for schema objects can be specified, but not the descriptions, since the descriptions cannot be overridden. Name cannot be overridden either, but each object needs a name for identification, so it must be specified.

The Schema Override Set is used to specify schema-wide overrides such as:

- Oracle Database for all tables for classes and object properties in the schema. Defaults to the current Oracle Database for the current connection.
- Oracle Owner for all tables for classes and object properties in the schema. Defaults to the current Oracle Schema for the current connection.
- Tablespace for all tables for classes and object properties in the schema. Defaults to the default table space for the Oracle Owner.
- Default table mapping type for all classes in the schema. If not specified, the default table mapping type is Concrete.

These schema-wide overrides can themselves be overridden on an element-by-element basis. For example, there are overrides available for class table, object property, and geometric property.

Class Table Overrides

The RDBMS table for storing class properties can be specified by adding a table to the class. The table specifies the table name and table primary key name. By default, the table name is set to be the same as the class name.

Data Property Overrides

The physical representation for a data property can be overridden by attaching a column to it. The column specifies the name of the property's corresponding column in the FDO database. If Column is not specified, then the column names default to Name (the property name).

Object Property Overrides

The type of an Object Property is a class in a Feature Schema. This class can be considered the referenced class. This referenced class has properties, so a home for each property must be provided in the RDBMS data store. There are two different ways to store these properties. The Mapping Definition for each Object Property is specified by setting its MappingDefinition to an object of one of the following classes:

- **PropertyMappingSingle.** The referenced class properties are embedded in the containing class's table. The containing class is the class containing the Object Property.
- **PropertyMappingConcrete.** The Object Property is not stored in the containing class's table. A separate table is automatically generated for it.

Geometric Property Overrides

The column for a Geometric property can be overridden by attaching a Geometric column to it. Only the column name can be specified. The column type must always be mdsys.sdo_geometry.

The default column depends on whether the F_Geometry_0 table is present and whether the Geometric Property is also the GeometryProperty for its containing class.

If it is the GeometryProperty and F_Geometry_0 exists, then the table for this property is F_Geometry_<n>, where <n> is the ID of the Spatial Context Group for the associated Spatial Context. If the Geometric Property is not associated with a Spatial Context, then <n> is the ID of the active Spatial Context group, at the time the geometric property is created. The column for the property is always RDBMS_GEOM.

Otherwise, the column name is assumed to be the same as the property name. The column table is assumed to be the table for the containing class.

Oracle-Specific Schema Creation Restrictions

This section describes the restrictions that apply when creating schema(s) using FDO Provider for Oracle.

FDOFeatureClass

An FdoFeatureClass must have an identity.

Classes

- Class names must be unique across the data store.
- FdoFeatureClass must define or inherit at least one IdentityProperty.

Properties

Restrictions apply to specific types of properties.

Data Properties

- The default value must not be specified.
- A non-nullable data property cannot be added to a class that already has data.

Identity Properties

- Identity properties cannot be nullable.
- Read-only Identity properties must be autogenerated.

String Properties

String property length must be between 1 and 4000 bytes inclusive.

Decimal Properties

- Decimal property precision must be between 0 and 38 inclusive.
- Decimal property scale must be between -84 and 127 inclusive.

Geometric Properties

- Only FdoFeatureClass can have geometric properties. A feature class can have multiple geometric properties; main geometry is not mandatory. HasMeasure and HasElevation are supported.
- If the geometric property values are stored in a feature geometry system table (F_GEOMETRY_<n>), then HasMeasure must be false.

Object Properties

- The object property class must be an FdoClass. (FdoFeatureClass is not allowed.)
- IdentityProperty is mandatory if ObjectType is not FdoObjectType_Value and the object property class has no identity properties.

Oracle-Specific Schema Modification Restrictions

This section describes restrictions that apply when modifying schema(s) using FDO Provider for Oracle.

Almost all modifications are disallowed, with the exception of those that follow.

Schema Element Descriptions

- Any schema element description is allowed.
- Any schema attribute dictionary (entries can be added, deleted, or modified) is allowed.

Data Properties

The read-only setting for a data property can be modified if the property is not autogenerated.

Oracle-Specific Deletion Restrictions

This section describes restrictions that apply when performing deletion in a schema while using FDO Provider for Oracle.

FDOClassDefinition

FdoClassDefinition cannot be deleted if it has data (objects).

FdoClassDefinition cannot be deleted if another class in the data store has it as its base class.

FDOClass

FdoClass cannot be deleted if it is referenced by any object property in the data store.

Property

- A data property cannot be deleted if it has any non-null values.
- An object property cannot be deleted if it has data.
- A geometric property cannot be deleted if its containing class has data.

Oracle Reserved Words Used with Filter and Expression Text

When using a filter string with Oracle reserved words, the string within the expression must be encapsulated inside single quotes (following the same convention used with the SQL language). Failure to do so will result in a parsing error because the parser cannot determine any difference between the value and the keyword.

Example of a filter string:

```
AND='linetype'
```

This FDO constraint applies to the Oracle reserved words:

- AND
- DATE
- IN
- LIKE
- NOT
- OR

Locking and Long Transactions

The purpose of this section is two-fold. First, it illustrates ways of understanding the subtleties of the interactions between locking and long transactions in an Oracle context. Secondly, it provides concrete examples of those subtleties.

An FDO long transaction version is called a workspace in an Oracle context. In this discussion, the FDO phrase “long transaction version” is shortened to “long transaction”. A key phrase in the example is “root,” which represents permanent data. Any long transaction has a root long transaction as an ancestor. The Oracle Workspace Manager (OWM) name for the FDO root long transaction is “LIVE”.

Version Enabling

The Autodesk FDO Provider for Oracle creates tables in the FDO data store that are not automatically version-enabled. Therefore, when you create a new Oracle data store using the default options, the resulting table is not version-enabled, so persistent locking and long transaction are not supported. (This differs from previous releases.)

OWM is used for versioning and persistent locking support. To enable versioning, you must execute the *EnableVersioning.sql* script in the */FDO/bin/com* folder. This will enable the tables for OWM. Use SQL*Plus to execute the scripts.

NOTE If you create a data store in AutoCAD Map 3D 2008 that you want to use with the previous version of Autodesk Map 3D, you must set the value of the lock and long transaction options in the table `F_Options` in the generated data store to 2. You can do this with the supplied SQL script *EnableVersioning.sql*, which also enables versioning for all tables and allows the creation of conditional data. Do not make this change to `F_Options` in the database if you do not plan to use it with the previous version of Autodesk Map 3D.

Read the documentation contained within the script files themselves to determine what privileges are required for each script, how to run the scripts, and what errors may occur. Severe consequences can occur if you respond incorrectly to any errors you encounter while running a script.

NOTE The *DisableVersioning.sql* script in the same folder provides the opposite functionality.

Before executing the scripts, the following conditions must be true:

- You always connect directly to the Oracle user (or FDO data store) to be processed.
- The Oracle user executing the script has sufficient privileges (this user has been granted the Workspace Manager role WM_ADMIN_ROLE).
- The Oracle user executing the script is the only user processing or accessing the current Oracle user (or FDO data store) during the execution of the script. Otherwise, a script failure may result from a session conflict.

You can create a script log file by executing the `spool <log file name>;` command before invoking the scripts and the `spool off;` command after the invoked script finishes. The log file can help you resolve any issues encountered by the scripts.

OWM and FDO Lock Types

The following table shows the names of the Oracle Workspace Manager locks used to implement each FDO lock:

FDO Lock Type	Oracle Lock Type
Shared	Shared
Exclusive	Workspace Exclusive
Long Transaction Exclusive	Version Exclusive
All Long Transaction Exclusive	Exclusive

Example: AllLongTransactionExclusiveLock

The following is a proven example using the AllLongTransactionExclusiveLock type with multiple users and the Update command. When you connect to an Oracle data store, you are placed in the already-activated, default root long transaction. If a long transaction is created in root, it is considered a child of root. When the new long transaction is activated (for example, as LT1), the

subsequent actions take place in the context of LT1. If another long transaction is subsequently created (for example, as LT2), it is created as a child of LT1.

NOTE When using FDO Provider for Oracle long transactions and locking, the combination of Oracle Workspace Manager capabilities and, potentially, other third-party applications can introduce many variables and combinations. The possible resulting conflicts in locking and long transactions can be similarly wide and varied.

This example considers two closely related cases. The same set of actions are taken in both cases, but in slightly different sequences, yielding different results. User1 creates a long transaction in the context of root and it is activated. User1 applies an AllLongTransactionExclusiveLock to a feature object in a data store. User1 updates that feature object in the data store. User2 attempts to update the same object, in the same data store, in the context of root. In the first case, User2 succeeds, and in the second case User2 fails (that is, a lock conflict is reported).

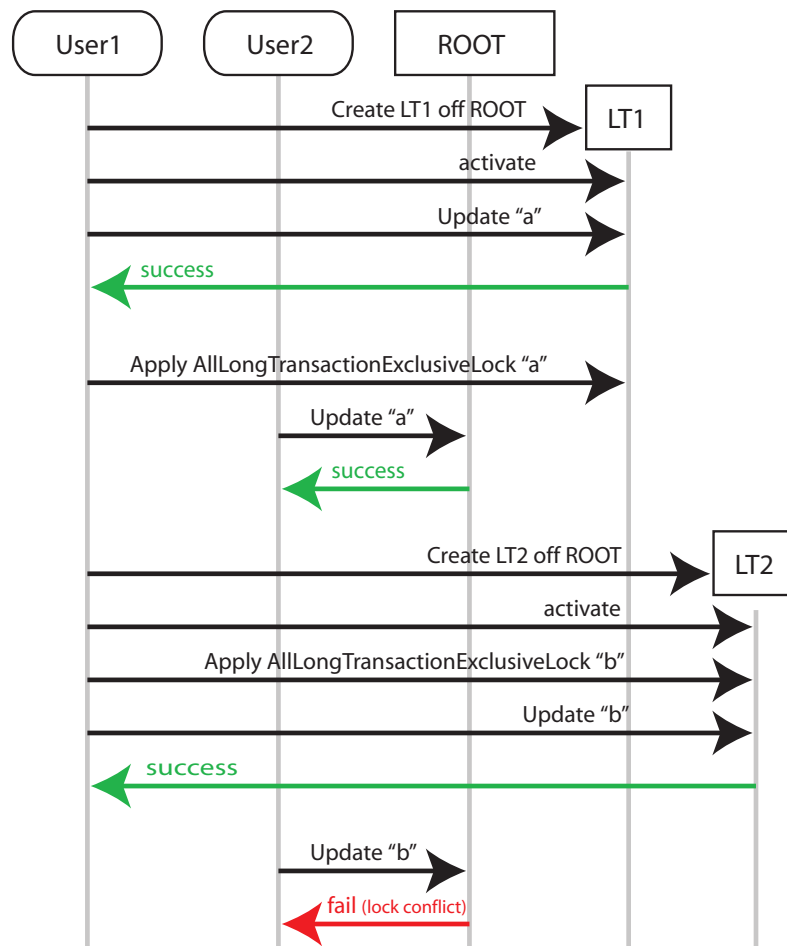
More specifically, the sequence of events for both cases is captured in the accompanying Long Transaction and Locking Sequencing Example diagram. For this example, all events occur in a single data store. The sequence of events are:

- 1 User1 creates LT1, activates LT1, and updates feature object “a” in LT1.
- 2 User2 successfully updates object “a” in root.
- 3 User1 creates LT2, activates LT2, and updates feature object “b” in LT2.
- 4 User2 fails to update object “b” in root.

The key difference is that, in LT1, User1 updates feature object “a” before the lock is applied, and, in LT2, User1 applies the lock to feature object “b” before it is updated. Prior to update, a copy of object “b” has not been made in LT2. This causes the lock to be applied to the copy of the object in root, because there is not yet a copy in LT2.

Therefore, if User1 intends to prevent anyone from modifying the object from the root level, User1 must apply the lock to the object *before* updating it.

For more information about Oracle Workspace Manager and its lock management, see the Oracle documentation.



Long Transaction and Locking Sequencing Example

FDO Provider for Oracle Capabilities

The capabilities of an FDO provider are grouped in the following categories:

- Connection
- Schema
- Commands

- Filters
- Expressions
- Geometry
- Raster

Connection Capabilities

Use the `FdoIConnectionCapabilities` object methods to learn about these capabilities. You can get this object by calling the `GetConnectionCapabilities()` method on the `FdoIConnection` object. For an explanation of the meaning of the capabilities, consult the `FdoIConnectionCapabilities` class description in the FDO API Reference documentation.

The following capabilities are supported:

- Per connection threading
- static spatial content extent type
- locking
- lock types: shared, exclusive, transaction, `AllLongTransactionExclusive`, `LongTransactionExclusive`
- transactions
- long transactions
- SQL
- XML configuration
- multiple spatial contexts
- specification of the coordinate system by name without specifying the WKT
- Write
- Multi-user write

Schema Capabilities

Use the `FdoISchemaCapabilities` object methods to learn about these capabilities. You can get this object by calling the `GetSchemaCapabilities()`

method on the `FdoIConnection` object. For an explanation of the meaning of the capabilities, consult the `FdoISchemaCapabilities` class description in the FDO API Reference documentation.

The following capabilities are supported:

- class and feature class class types
- Boolean data type with a maximum length of 1 byte
- Byte data type with a maximum length of 1 byte
- DateTime data type with a maximum length of 12 bytes
- Decimal data type with a maximum length of 165 digits (maximum decimal precision of 38 and maximum decimal scale of 127)
- Double data type with a maximum length of 8 bytes
- Int16 data type with a maximum length of 2 bytes
- Int32 data type with a maximum length of 4 bytes
- Int64 data type with a maximum length of 8 bytes
- Single data type with a maximum length of 4 bytes
- String data type with a maximum length of 4000
- BLOB data type with a maximum length of 4000000000 bytes
- CLOB data type with a maximum length of 4000000000 bytes
- Int64 auto-generated data type
- Identity properties of type Boolean, Byte, DateTime, Decimal, Double, Int16, Int32, Int64, Single, and String
- Name size limitation of 30 for a schema element name of type `FdoSchemaElementNameType_Datastore`
- Name size limitation of 255 for a schema element name of type `FdoSchemaElementNameType_Schema`
- Name size limitation of 255 for a schema element name of type `FdoSchemaElementNameType_Class`
- Name size limitation of 255 for a schema element name of type `FdoSchemaElementNameType_Property`

- Name size limitation of 255 for a schema element name of type `FdoSchemaElementType_Description`
- Characters that cannot be used for a schema element name: `..`
- Association properties
- Auto ID generation
- Composite ID
- Composite unique value constraints
- Datastore scope unique ID generation
- Default value
- Exclusive value range constraints
- Inclusive value range constraints
- Inheritance
- Multiple schemas
- Null value constraints
- Object properties
- Unique value constraints
- Schema modification
- Schema overrides
- Unique value constraints
- Value constraints list

Command Capabilities

Use the `FdoICommandCapabilities` object methods to learn about these capabilities. You can get this object by calling the `GetCommandCapabilities()` method on the `FdoIConnection` object. For an explanation of the meaning of the capabilities, consult the `FdoICommandCapabilities` class description in the FDO API Reference documentation.

The following commands are supported:

- `FdoCommandType_Select`

- FdoCommandType_SelectAggregates
- FdoCommandType_Insert
- FdoCommandType_Delete
- FdoCommandType_Update
- FdoCommandType_DescribeSchema
- FdoCommandType_DescribeSchemaMapping
- FdoCommandType_ApplySchema
- FdoCommandType_DestroySchema
- FdoCommandType_ActivateSpatialContext
- FdoCommandType_CreateSpatialContext
- FdoCommandType_DestroySpatialContext
- FdoCommandType_GetSpatialContexts
- FdoCommandType_CreateDataStore
- FdoCommandType_DestroyDataStore
- FdoCommandType_ListDataStores
- FdoCommandType_SQLCommand
- FdoCommandType_AcquireLock
- FdoCommandType_GetLockInfo
- FdoCommandType_GetLockedObjects
- FdoCommandType_GetLockOwners
- FdoCommandType_ReleaseLock
- FdoCommandType_ActivateLongTransaction
- FdoCommandType_CommitLongTransaction
- FdoCommandType_CreateLongTransaction
- FdoCommandType_DeactivateLongTransaction
- FdoCommandType_GetLongTransactions
- FdoCommandType_RollbackLongTransaction

- FdoRdbmsCommandType_CreateSpatialIndex
- FdoRdbmsCommandType_DestroySpatialIndex
- FdoRdbmsCommandType_GetSpatialIndexes

The following capabilities are supported:

- simple functions in Select and SelectAggregate commands
- use of expressions for properties in Select and SelectAggregates commands
- use of Distinct in SelectAggregates command
- availability of ordering in Select and SelectAggregates command
- availability of grouping criteria in SelectAggregates command

Filter Capabilities

Use the FdoIFilterCapabilities object methods to learn about these capabilities. You can get this object by calling the GetFilterCapabilities() method on the FdoIConnection object. For an explanation of the meaning of the capabilities, consult the FdoIFilterCapabilities class description in the FDO API Reference documentation.

The following capabilities are supported:

- Conditions of type comparison, like, in, null, spatial, and distance
- the Within distance operation
- spatial operations of type CoveredBy, Inside, Intersects, EnvelopeIntersects

Expression Capabilities

Use the FdoIExpressionCapabilities object methods to learn about these capabilities. You can get this object by calling the GetExpressionCapabilities() method on the FdoIConnection object. For an explanation of the meaning of the capabilities, consult the FdoIExpressionCapabilities class description in the FDO API Reference documentation.

Basic, function, and parameter expressions are supported.

The following functions are supported:

- Double Avg(<type> value) where <type> is one of Decimal, Double, Single, Int16, Int32, or Int64.
- Decimal Ceil(<type> value) where <type> is one of Decimal, Double, or Single
- String Concat(String str1Val, String str2Val)
- Int64 Count(<type> value) where <type> is one of Boolean, Byte, DateTime, Decimal, Double, Int16, Int32, Int64, Single, String, BLOB, or CLOB
- Decimal Floor(<type> value) where <type> is one of Decimal, Double, or Single
- String Lower(String value)
- Byte Max(<type> value) where <type> is one of Byte, DateTime, Decimal, Double, Int16, Int32, Int64, Single, or String.
- Byte Min(<type> value) where <type> is one of Byte, DateTime, Decimal, Double, Int16, Int32, Int64, Single, or String.
- Double Sum(<type> value) where <type> is one of Decimal, Double, Int16, Int32, Int64, or Single.
- String Upper(String value)
- GeometricProperty SpatialExtents(GeometricProperty property)

Geometry Capabilities

Use the FdoGeometryCapabilities object methods to learn about these capabilities. You can get this object by calling the GetGeometryCapabilities() method on the FdoIConnection object. For an explanation of the meaning of the capabilities, consult the FdoGeometryCapabilities class description in the FDO API Reference documentation.

Dimensionality XYZM is supported. The geometry component types Ring, LinearRing, CircularArcSegment, and LineStringSegment are supported. The following geometry types are supported.

- Point
- LineString
- Polygon

- MultiPoint
- MultiLineString
- MultiPolygon
- CurveString
- CurvePolygon
- MultiCurveString
- MultiCurvePolygon

Raster Capabilities

Use the `FdoIRasterCapabilities` object methods to learn about these capabilities. You can get this object by calling the `GetRasterCapabilities()` method on the `FdoIConnection` object. For an explanation of the meaning of the capabilities, consult the `FdoIRasterCapabilities` class description in the FDO API Reference documentation.

No Raster capabilities are supported.

OSGeo FDO Provider for ArcSDE

B

This appendix discusses FDO API development issues that are related to OSGeo FDO Provider for ArcSDE.

What Is FDO Provider for ArcSDE?

The Feature Data Objects (FDO) API provides access to data in a data store. A provider is a specific implementation of the FDO API that provides access to data in a particular data store. A provider is a specific implementation of the FDO API that provides access to data in a particular data store. ESRI® ArcSDE® (Spatial Database Engine) is part of the ArcGIS 9 system. ArcSDE manages the exchange of information between an (ArcGIS 9 Desktop) application and a relational database management system. FDO Provider for ArcSDE provides FDO with access to an ArcSDE 9-based data store, which, in this case, must be Oracle 9i (9.2.0.6).

FDO Provider for ArcSDE Software Requirements

Installed Components

FDO Provider for ArcSDE dynamically linked libraries are installed with the FDO SDK. They are located in <FDO SDK Install Location>\FDO\bin. You do not have to do anything to make these DLLs visible.

External Dependencies

The operation of FDO Provider for ArcSDE is dependent on the presence of ArcSDE 9 and a supported data source, such as Oracle 9i, in the network environment. The host machine running FDO Provider for ArcSDE must also have the required DLLs present, which are available by installing either an ArcGIS 9.1 Desktop application or the ArcSDE SDK. For example, the required DLLs are present if either ArcView®, ArcEditor®, or ArcInfo® are installed. For more information about ArcGIS 9.1 Desktop applications and the ArcSDE SDK, refer to the ESRI documentation.

Specifically, in order for FDO Provider for ArcSDE to run, three dynamically linked libraries, sde91.dll, sg91.dll, and pe91.dll, are required and you must ensure that the PATH environment variable references the local folder containing these DLLs. For example, in Microsoft Windows, if ArcGIS 9.1 Desktop is installed to C:\Program Files\ArcGIS, then the required ArcSDE binaries are located at C:\Program Files\ArcGIS\ArcSDE\bin. Similarly, if the ArcSDE SDK is installed to the default location, then the required ArcSDE binaries are located at C:\ArcGis\ArcSDE\bin. The absence of this configuration may cause the following exception message "The ArcSDE runtime was not found."

FDO Provider for ArcSDE Limitations

The FDO Provider for ArcSDE is based on a subset of the ArcSDE API. This subset does not include the following:

- Raster functionality
- Native ArcSDE metadata
- The annotation data, with the exception of the ANNO_TEXT column

ArcSDE Limitations

FDO Provider for ArcSDE must abide by limitations of the ArcSDE technology to which it connects. This section discusses these limitations.

Relative to ArcObjects API and ArcGIS Server API

The ArcSDE API does not support the following advanced functionality found in the ArcObjects API and the newer ArcGIS Server API:

- Advanced geometries, such as Bezier curves and ellipses
- Relationships
- Topology
- Networks
- Analysis
- Linear referencing

Curved Segments

If ArcSDE encounters curved segments, it will automatically tessellate them. This means that if you create a geometry containing an arc segment in an ArcSDE table using ArcObjects API and then you try to read that geometry back using the ArcSDE API, you will get a series of line segments that approximate the original arc segment. That is, you get an approximation of the original geometry.

Locking and Versioning

ArcSDE permits row locks or table versioning provided that the ID column, which uniquely identifies the row, is maintained by ArcSDE. If there is no ID column or the ID column is maintained by the user, ArcSDE does not permit row locking or table versioning to be enabled.

NOTE In ArcSDE you can either lock rows in a table or version a table, but you cannot do both at the same time. To do either, you must alter the table's registration.

The following sections illustrate these three steps:

- 1 The creation of a table.

- 2 The alteration of the table registration to identify one of the column definitions as the row ID column and to enable row locking.
- 3 The alteration of the table registration to disable row locking and to enable versioning.

Table Creation

The command is:

```
sddtable -o create -t hassdemaintainedrowid -d "name string(20),
fid integer(9)" -u t_user -p test
```

The output of the describe registration command (sddtable -o describe_reg) for this table is as follows:

NOTE The Row Lock has no value and the value of Dependent Objects is None.

Table Owner	: T_USER
Table Name	: HASSEMAINTAINEDROWID
Registration Id	: 18111
Row ID Column	:
Row ID Column Type	:
Row Lock	:
Minimum Row ID	:
Dependent Objects	: None
Registration Date	: 02/24/05 13:08:02
Config. Keyword	: DEFAULTS
User Privileges	: SELECT, UPDATE, INSERT, DELETE
Visibility	: Visible

Identity Row ID Column and Enable Row Locking

The command is:

```
sddtable -o alter_reg -t hassdemaintainedrowid -c fid -C sde -L
on -u t_user -p test
```

The output of the describe registration command (sddtable -o describe_reg) for this table is as follows.

NOTE The Row ID Column value is FID, the Row ID Column Type value is SDE Maintained, and the Row Lock value is Enable.

Table Owner	: T_USER
Table Name	: HASSEMAINTAINEDROWID
Registration Id	: 18111
Row ID Column	: FID
Row ID Column Type	: SDE Maintained
Row ID Allocation	: Many
Row Lock	: Enable
Minimum Row ID	: 1
Dependent Objects	: None
Registration Date	: 02/24/05 13:08:02
Config. Keyword	: DEFAULTS
User Privileges	: SELECT, UPDATE, INSERT, DELETE
Visibility	: Visible

Disable Row Locking and Enable Versioning

The command is:

```
sdetable -o alter_reg -t hassdemaintainedrowid -L off -V MULTI -u  
t_user -p test
```

The output of the describe registration command (sdetable -o describe_reg) for this table is as follows:

NOTE The “Row Lock” is “Not Enable” and “Dependent Objects” is “Multiversion Table”.

Table Owner	: T_USER
Table Name	: HASSEMAINTAINEDROWID
Registration Id	: 18111
Row ID Column	: FID
Row ID Column Type	: SDE Maintained
Row ID Allocation	: Many
Row Lock	: Not Enable
Minimum Row ID	: 1
Dependent Objects	: Multiversion Table
Dependent Object Names	: A18111, D18111
Registration Date	: 02/24/05 13:08:02
Config. Keyword	: DEFAULTS
User Privileges	: SELECT, UPDATE, INSERT, DELETE
Visibility	: Visible

FDO Provider for ArcSDE Connection

This information supplements the Establishing a Connection chapter. You connect to an ArcSDE data store indirectly through the ArcSDE server. The underlying data source for the data store is a database, such as Oracle. The ArcSDE server is connected to the data source and mediates the requests that you send it.

You can connect to FDO Provider for ArcSDE in one step if you already know the name of the data store that you want to use. Otherwise, you must connect in two steps.

The minimum required connection properties for the initial call to Open() are server, instance, username, and password. Multiple users can access the data store. However, access is password-protected. The server property is the name of the machine hosting the ArcSDE server. The instance property acts as an index into an entry in the services file. An entry contains port and protocol information used to connect to the ArcSDE server. On a Windows machine, the services file is located in C:\WINDOWS\system32\drivers\etc. Assuming that the instance name is "esri_sde", an entry would look something like this: "esri_sde 5151/tcp #ArcSDE Server Listening Port".

An ArcSDE data source may contain more than one data store. For the first call to Open(), a data store name is optional. If successful, the first call to Open() results in the data store parameter becoming a required parameter and a list of the names of the data stores in the data source becoming available. You must choose a data store and call Open() again.

If the data source supports multiple data stores, the list returned by the first call to `Open()` will contain a list of all of the data stores resident in the data source. Otherwise, the list will contain one entry: “Default Data Store”.

If you know the name of the data store, you can provide it for the first call to `Open()` and make the connection in one step.

Data Type Mappings

This section shows the mappings from FDO data types to ArcSDE data types to Oracle data types:

FDO DataType	sdetable Column Definition	Oracle Column Type
FdoDataType_Boolean	Not supported	Not supported
FdoDataType_Byte	Not supported	Not supported
FdoDataType_DateTime	date	DATE
FdoDataType_Decimal	Not supported	Not supported
FdoDataType_Double	double(38,8)	NUMBER(38,8)
FdoDataType_Int16	integer(4)	NUMBER(4)
FdoDataType_Int32	integer(10)	NUMBER(10)
FdoDataType_Int64	Not supported	Not supported
FdoDataType_Single	float(6,2) // typical float(0<n<=6, 0<m<DBMSLimit)) // possible	NUMBER(6,2) NUMBER(n,8)
FdoDataType_String	string(<length>)	VARCHAR2(<length>)
FdoDataType_BLOB	blob	LONG RAW
FdoDataType_CLOB	Not supported	Not supported

FDO DataType	shtable Column Definition	Oracle Column Type
FdoDatatype_UniqueID	Not supported	Not supported

Creating a Feature Schema

This section describes the creation of the SampleFeatureSchema, which is the example feature schema described in the Schema Management chapter. It also describes the creation of the OGC980461FS schema, which is the schema defined in the OpenGIS project document 98-046r1.

FDO Provider for ArcSDE does not support the creation or destruction of feature schema (that is, does not support the FdoIApplySchema and FdoIDestroySchema commands.) However, it does support the FdoIDescribeSchema command. The intended use of FDO Provider for ArcSDE is to operate on already existing feature schemas. FDO Provider for ArcSDE supports inserting, selecting, updating, and deleting data in existing schemas.

You can use FDO Provider for ArcSDE to operate on a new feature schema. However, you must create the schema using ArcSDE tools. In particular you use the shtable and sdelayer commands, which can be used to create a schema in any of the data store technologies used by ArcSDE. This part of the description is generic. Other parts of the description are specific to Oracle and to Windows XP because Oracle is the data store technology and Windows XP is the operating system for this exercise.

First, you must create an Oracle username for the feature schema (that is, the name of the Oracle user is the name of the feature schema.) To do this, you connect as system administrator to the Oracle instance used by the ArcSDE server. The following command creates the user and grants to that user the privileges necessary for the ArcSDE tool commands to succeed:

```
grant connect,resource to <schemaName> identified by <password>
```

Secondly, you must log in to the host where the ArcSDE server is running. ArcSDE tools are on the host machine where the ArcSDE server resides.

TIP NetMeeting can be used to remotely login to where the ArcSDE Server is running and launch a command window (that is, in the Run dialog box, enter cmd) The ArcSDE tool commands can be executed through the command window. Do not use C:\WINDOWS\SYSTEM32\COMMAND.COM because the line buffer is too short to contain the entire text of some of the SDE tool command strings.

Finally, execute the `shtable` and `sdlayer` commands in a command window to create each of the classes. Since you are executing these commands on the host where the ArcSDE server is located, you can omit the server name option. If the ArcSDE server is connected to only one data store, you can omit the service option. For more information about all of the ArcSDE commands, consult the ArcSDE Developer Help Guide.

SampleFeatureSchema

In this sample a feature schema called `SampleFeatureSchema` is created, which contains one feature class called `SampleFeatureClass`. This feature class has the following three properties:

- An `Int32` called `SampleIdentityDataProperty`.
- A string called `SampleNameDataProperty`.
- A polygon geometry called `SampleGeometricProperty`.

First, use the `shtable -o create` command to add the integer and string properties to `SampleFeatureClass`. Then, use the `shtable -o alter_reg` command to identify the `SampleIdentityDataProperty` as an identity property. Finally, use the `sdlayer -o add` command to add the geometric property to `SampleFeatureClass`. This assumes that only one ArcSDE server service is running so that the `-i` option is optional. The `-i` option takes a service name as an argument.

The `shtable -o create` command can be invoked as follows:

```
shtable -o create -t SampleFeatureClass -d "SampleIdentityData
Property INTEGER(10), SampleNameDataProperty STRING(64)" -u
SampleFeatureSchema -p test.
```

The `-o` option takes the command option name. The `-d` option takes the column definitions, which is a quoted list of column name/column type pairs delimited by commas. The `-u` option takes an Oracle database user name, which becomes the feature schema name. The `-p` option takes a password.

The `shtable -o alter_reg` command is invoked as follows:

```
shtable -o alter_reg -t SampleFeatureClass -c SampleIdentityData
Property -C USER -u SampleFeatureSchema -p test
```

The `-c` option identifies the column name that will be the identity property. The `-C` option indicates whether SDE is supposed to generate the value or obtain it from the user. You will be prompted to confirm that you want to alter the registration of the table.

The `sdelayer` command is invoked as follows:

```
sdelayer -o add -l SampleFeatureClass,SampleGeometricProperty -E  
0,0,100,50 -e a -u SampleFeatureSchema -p test
```

The `-o` option takes the command option name. The `-l` option identifies the table and column. The `-E` option identifies the extents; the arguments are `<xmin,ymin,xmax,ymax>`. The `-e` option identifies the geometry type with 'a' indicating an area shape.

OGC98046 IFS

This schema contains the ten classes defined in the OpenGIS Project Document *980946r1*. The types of the properties belonging to the classes is similar to that of `SampleFeatureClass`, namely, an integer, a string, and a geometry. One difference is that the geometry in three of the classes is multipart. Two of them have MULTIPOLYGON geometries, and one of them has a MULTILINESTRING geometry. A multipart geometry is indicated by adding a '+' to the entity argument to the `-e` option in the `sdelayer` command. A MULTIPOLYGON geometry is indicated by "`-e a+`", and a MULTILINESTRING geometry is indicated by "`-e l+`".

An ArcSDE table cannot have two geometries. This restriction impacts the definition of the buildings class, which has a POLYGON and a POINT geometry. We have chosen to add the POINT geometry. The OpenGIS 98-046r1 document defines one query that references building objects, and the POINT geometry supports this query.

NOTE The use of `-E` option in the `sdelayer` command defines the extents. The arguments are `<xmin,ymin,xmax,ymax>`. The values provided below ensure that you will not receive any "ordinate out of bounds" errors when inserting the 98046r1 data.

ArcSDE Commands That Define the OGC98046 IFS Classes

```

sddtable -o create -t lakes -d "fid integer(10), name string(64)"
-u OGC980461FS -p test
sddtable -o alter_reg -t lakes -c fid -C user -u OGC980461FS -p
test
sddlayer -o add -l lakes,shore -E 0,0,100,50 -e a -u OGC980461FS
-p test
sddtable -o create -t road_segments -d "fid integer(10), name
string(64), aliases string(64), num_lanes integer(10)" -u
OGC980461FS -p test
sddtable -o alter_reg -t road_segments -c fid -C user -u
OGC980461FS -p test
sddlayer -o add -l road_segments,centerline -E 0,0,100,50 -e l -u
OGC980461FS -p test
sddtable -o create -t divided_routes -d "fid integer(10), name
string(64), num_lanes integer(10)" -u OGC980461FS -p test
sddtable -o alter_reg -t divided_routes -c fid -C user -u
OGC980461FS -p test
sddlayer -o add -l divided_routes,centerlines -E 0,0,100,50 -e l+
-u OGC980461FS -p test
sddtable -o create -t forests -d "fid integer(10), name string(64)"
-u OGC980461FS -p test
sddtable -o alter_reg -t forests -c fid -C user -u OGC980461FS -p
test
sddlayer -o add -l forests,boundary -E 0,0,100,50 -e a+ -u
OGC980461FS -p test
sddtable -o create -t bridges -d "fid integer(10), name string(64)"
-u OGC980461FS -p test
sddtable -o alter_reg -t bridges -c fid -C user -u OGC980461FS -p
test
sddlayer -o add -l bridges,position -E 0,0,100,50 -e p -u
OGC980461FS -p test
sddtable -o create -t streams -d "fid integer(10), name string(64)"
-u OGC980461FS -p test
sddtable -o alter_reg -t streams -c fid -C user -u OGC980461FS -p
test
sddlayer -o add -l streams,centerline -E 0,0,100,50 -e l -u
OGC980461FS -p test
sddtable -o create -t buildings -d "fid integer(10), address
string(64)" -u OGC980461FS -p test
sddtable -o alter_reg -t buildings -c fid -C user -u OGC980461FS
-p test
sddlayer -o add -l buildings,position -E 0,0,100,50 -e p -u
OGC980461FS -p test

```

```

sddtable -o create -t ponds -d "fid integer(10), name string(64),
  type string(64)" -u OGC980461FS -p test
sddtable -o alter_reg -t ponds -c fid -C user -u OGC980461FS -p
test
sddlayer -o add -l ponds,shores -E 0,0,100,50 -e a+ -u OGC980461FS
-p test
sddtable -o create -t named_places -d "fid integer(10), name
string(64)" -u OGC980461FS -p test
sddtable -o alter_reg -t named_places -c fid -C user -u OGC980461FS
-p test
sddlayer -o add -l named_places,boundary -E 0,0,100,50 -e a -u
OGC980461FS -p test
sddtable -o create -t map_neatlines -d "fid integer(10)" -u
OGC980461FS -p test
sddtable -o alter_reg -t map_neatlines -c fid -C user -u
OGC980461FS -p test
sddlayer -o add -l map_neatlines,neatline -E 0,0,100,50 -e a -u
OGC980461FS -p test

```

FDO Provider for ArcSDE Capabilities

The capabilities of an FDO provider are grouped in the following categories:

- Connection
- Schema
- Commands
- Expressions
- Filters
- Geometry
- Raster

Connection Capabilities

Use the `FdoIConnectionCapabilities` object methods to learn about these capabilities. You can get this object by calling the `GetConnectionCapabilities()` method on the `FdoIConnection` object. For an explanation of the meaning of the capabilities, consult the `FdoIConnectionCapabilities` class description in the FDO API Reference documentation.

The following capabilities are supported:

- Per connection threading
- static spatial content extent type
- locking
- exclusive locking type
- transactions
- long transactions
- SQL
- multiple spatial contexts
- specifying coordinate systems by name or ID without specifying WKT
- Write
- Multi-user write

Schema Capabilities

Use the `FdoSchemaCapabilities` object methods to learn about these capabilities. You can get this object by calling the `GetSchemaCapabilities()` method on the `FdoIConnection` object. For an explanation of the meaning of the capabilities, consult the `FdoSchemaCapabilities` class description in the FDO API Reference documentation.

The following capabilities are supported:

- class and feature class class types
- `DateTime` data type with a maximum length of 12 bytes
- `Double` data type with a maximum length of 8 bytes
- `Int16` data type with a maximum length of 2 bytes
- `Int32` data type with a maximum length of 4 bytes
- `Single` data type with a maximum length of 4 bytes
- `String` data type with a maximum length of 4294967296
- `BLOB` data type with a maximum length of 4294967296
- `Int32` auto-generated data type

- Identity properties of type DateTime, Double, Int16, Int32, Single, String, and BLOB
- Name size limitation of 123 for a schema element name of type FdoSchemaElementNameType_Datastore
- Name size limitation of 65 for a schema element name of type FdoSchemaElementNameType_Schema
- Name size limitation of 160 for a schema element name of type FdoSchemaElementNameType_Class
- Name size limitation of 32 for a schema element name of type FdoSchemaElementNameType_Property
- Name size limitation of 64 for a schema element name of type FdoSchemaElementNameType_Description
- Characters that cannot be used for a schema element name: ::
- Auto ID generation
- Composite unique value constraints
- Multiple schemas
- Null value constraints
- Unique value constraints

Command Capabilities

Use the FdoICommandCapabilities object methods to learn about these capabilities. You can get this object by calling the GetCommandCapabilities() method on the FdoIConnection object. For an explanation of the meaning of the capabilities, consult the FdoICommandCapabilities class description in the FDO API Reference documentation.

The following commands are supported:

- FdoCommandType_Select
- FdoCommandType_SelectAggregates
- FdoCommandType_Insert
- FdoCommandType_Delete
- FdoCommandType_Update

- FdoCommandType_DescribeSchema
- FdoCommandType_ActivateSpatialContext
- FdoCommandType_CreateSpatialContext
- FdoCommandType_DestroySpatialContext
- FdoCommandType_GetSpatialContexts
- FdoCommandType_SQLCommand
- FdoCommandType_AcquireLock
- FdoCommandType_GetLockInfo
- FdoCommandType_GetLockedObjects
- FdoCommandType_GetLockOwners
- FdoCommandType_ReleaseLock
- FdoCommandType_ActivateLongTransaction
- FdoCommandType_DeactivateLongTransaction
- FdoCommandType_CommitLongTransaction
- FdoCommandType_CreateLongTransaction
- FdoCommandType_GetLongTransactions
- FdoCommandType_RollbackLongTransaction
- FdoCommandType_ListDataStores

The following capabilities are supported:

- command parameterization
- simple functions in Select and SelectAggregate commands
- use of Distinct in SelectAggregates command

Filter Capabilities

Use the FdoIFilterCapabilities object methods to learn about these capabilities. You can get this object by calling the GetFilterCapabilities() method on the FdoIConnection object. For an explanation of the meaning of the capabilities,

consult the `FdoIFilterCapabilities` class description in the FDO API Reference documentation.

The following capabilities are supported:

- Conditions of type comparison, like, in, null, spatial, and distance
- the Beyond and Within distance operations
- spatial operations of type Contains, Crosses, Disjoint, Equals, Intersects, Overlaps, Touches, Within, CoveredBy, Inside, and EnvelopeIntersects

Expression Capabilities

Use the `FdoIExpressionCapabilities` object methods to learn about these capabilities. You can get this object by calling the `GetExpressionCapabilities()` method on the `FdoIConnection` object. For an explanation of the meaning of the capabilities, consult the `FdoIExpressionCapabilities` class description in the FDO API Reference documentation.

Basic expressions are supported.

The following functions are supported:

- `Double Sum(<type> value)` where <type> is one of Double, Single, Int16, or Int32.
- `Int64 Count(<type> value)` where <type> is one of Boolean, Double, Single, Decimal, Byte, DateTime, Int16, Int32, Int64, String, BLOB, CLOB, ObjectProperty, GeometricProperty, AssociationProperty, or RasterProperty
- `Double Avg(<type> value)` where <type> is one of Double, Single, Int16, or Int32.
- `Double Max(<type> value)` where <type> is one of Double, Single, Int16, or Int32.
- `Double StdDev(<type> value)` where <type> is one of Double, Single, Int16, or Int32.

Geometry Capabilities

Use the `FdoIGeometryCapabilities` object methods to learn about these capabilities. You can get this object by calling the `GetGeometryCapabilities()` method on the `FdoIConnection` object. For an explanation of the meaning of the capabilities, consult the `FdoIGeometryCapabilities` class description in the FDO API Reference documentation.

Dimensionality XYZM is supported. The geometry component types LinearRing and LineStringSegment are supported. The following geometry types are supported.

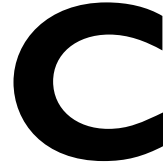
- Point
- LineString
- Polygon
-
- MultiPoint
- MultiLineString
- MultiPolygon

Raster Capabilities

Use the FdoIRasterCapabilities object methods to learn about these capabilities. You can get this object by calling the GetRasterCapabilities() method on the FdoIConnection object. For an explanation of the meaning of the capabilities, consult the FdoIRasterCapabilities class description in the FDO API Reference documentation.

No Raster capabilities are supported.

OSGeo FDO Provider for MySQL



This appendix discusses FDO API development issues that are related to OSGeo FDO Provider for MySQL.

What Is FDO Provider for MySQL?

The Feature Data Objects (FDO) API provides access to data in a data store. A provider is a specific implementation of the FDO API that provides access to data in a particular data store. The FDO Provider for MySQL provides FDO with access to a MySQL-based data store.

The FDO Provider for MySQL API provides custom commands that are specifically designed to work with the FDO API. For example, using these commands, you can do the following:

- Gather information about a provider.
- Transmit client services exceptions.
- Get lists of accessible data stores.
- Create connection objects.
- Create and execute spatial queries.

The MySQL architecture supports different storage engines. Choose an engine as needed, depending on its characteristics and capabilities, such as the following:

- MyISAM is a disk-based storage engine. It does not support transactions.
- InnoDB is a disk-based storage engine. It has full ACID transaction capability.

- Memory (Heap) is a storage engine utilizing only RAM. It is very fast.
- NDB is the MySQL Cluster storage engine.
- MERGE is a variation of MyISAM. A MERGE table is a collection of identical MyISAM tables, which means that all tables have the same columns, column types, indexes, and so on.

For more information, see *The Essential FDO* (FET_TheEssentialFDO.pdf) and the *OSGeo FDO Provider for MySQL API Reference Help* (MySQL_Provider_API.chm).

FDO Provider for MySQL Capabilities

The capabilities of an FDO provider are grouped in the following categories:

- Connection
- Schema
- Commands
- Expressions
- Filters
- Geometry
- Raster

Connection Capabilities

Use the `FdoIConnectionCapabilities` object methods to learn about these capabilities. You can get this object by calling the `GetConnectionCapabilities()` method on the `FdoIConnection` object. For an explanation of the meaning of the capabilities, consult the `FdoIConnectionCapabilities` class description in the FDO API Reference documentation.

The following capabilities are supported:

- Per connection threading
- static spatial content extent type
- transactions

- SQL
- multiple spatial contexts
- Write
- Multi-user write

Schema Capabilities

Use the `FdoSchemaCapabilities` object methods to learn about these capabilities. You can get this object by calling the `GetSchemaCapabilities()` method on the `FdoIConnection` object. For an explanation of the meaning of the capabilities, consult the `FdoSchemaCapabilities` class description in the FDO API Reference documentation.

The following capabilities are supported:

- class and feature class class types
- Boolean data type with a maximum length of 1 byte
- Byte data type with a maximum length of 1 byte
- DateTime data type with a maximum length of 12 bytes
- Decimal data type with a maximum length of 95 digits (maximum decimal precision of 65 and maximum decimal scale of 30)
- Double data type with a maximum length of 8 bytes
- Int16 data type with a maximum length of 2 bytes
- Int32 data type with a maximum length of 4 bytes
- Int64 data type with a maximum length of 8 bytes
- Single data type with a maximum length of 4 bytes
- String data type with a maximum length of 2147483647
- Int64 auto-generated data type
- Identity properties of type Boolean, Byte, DateTime, Decimal, Double, Int16, Int32, Int64, Single, and String
- Name size limitation of 64 for a schema element name of type `FdoSchemaElementNameType_Datastore`

- Name size limitation of 200 for a schema element name of type `FdoSchemaElementNameType_Schema`
- Name size limitation of 200 for a schema element name of type `FdoSchemaElementNameType_Class`
- Name size limitation of 255 for a schema element name of type `FdoSchemaElementNameType_Property`
- Name size limitation of 255 for a schema element name of type `FdoSchemaElementNameType_Description`
- Characters that cannot be used for a schema element name: `::`
- Association properties
- Auto ID generation
- Composite ID
- Composite unique value constraints
- Datastore scope unique ID generation
- Default value
- Exclusive value range constraints
- Inclusive value range constraints
- Inheritance
- Multiple schemas
- Null value constraints
- Object properties
- Unique value constraints
- Schema modification
- Schema overrides
- Unique value constraints

Command Capabilities

Use the `FdoICommandCapabilities` object methods to learn about these capabilities. You can get this object by calling the `GetCommandCapabilities()`

method on the `FdoIConnection` object. For an explanation of the meaning of the capabilities, consult the `FdoICommandCapabilities` class description in the FDO API Reference documentation.

The following commands are supported:

- `FdoCommandType_Select`
- `FdoCommandType_SelectAggregates`
- `FdoCommandType_Insert`
- `FdoCommandType_Delete`
- `FdoCommandType_Update`
- `FdoCommandType_DescribeSchema`
- `FdoCommandType_ApplySchema`
- `FdoCommandType_DestroySchema`
- `FdoCommandType_CreateSpatialContext`
- `FdoCommandType_DestroySpatialContext`
- `FdoCommandType_GetSpatialContexts`
- `FdoCommandType_CreateDataStore`
- `FdoCommandType_DestroyDataStore`
- `FdoCommandType_ListDataStores`
- `FdoCommandType_DescribeSchemaMapping`
- `FdoCommandType_SQLCommand`
- `FdoRdbmsCommandType_CreateSpatialIndex`
- `FdoRdbmsCommandType_DestroySpatialIndex`
- `FdoRdbmsCommandType_GetSpatialIndexes`

The following capabilities are supported:

- simple functions in `Select` and `SelectAggregate` commands
- use of expressions for properties in `Select` and `SelectAggregates` commands
- use of `Distinct` in `SelectAggregates` command

- availability of ordering in Select and SelectAggregates command
- availability of grouping criteria in SelectAggregates command

Filter Capabilities

Use the `FdoIFilterCapabilities` object methods to learn about these capabilities. You can get this object by calling the `GetFilterCapabilities()` method on the `FdoIConnection` object. For an explanation of the meaning of the capabilities, consult the `FdoIFilterCapabilities` class description in the FDO API Reference documentation.

The following capabilities are supported:

- Conditions of type comparison, like, in, null, spatial, and distance
- spatial operations of type `Intersects` and `EnvelopeIntersects`

Expression Capabilities

Use the `FdoIExpressionCapabilities` object methods to learn about these capabilities. You can get this object by calling the `GetExpressionCapabilities()` method on the `FdoIConnection` object. For an explanation of the meaning of the capabilities, consult the `FdoIExpressionCapabilities` class description in the FDO API Reference documentation.

Basic, function, and parameter expressions are supported.

The following functions are supported:

- `Double Avg(<type> value)` where `<type>` is one of `Decimal`, `Double`, `Single`, `Int16`, `Int32`, or `Int64`.
- `Decimal Ceil(<type> value)` where `<type>` is one of `Decimal` or `Double`.
- `String Concat(String str1Val, String str2Val)`
- `Int64 Count(<type> value)` where `<type>` is one of `Boolean`, `Byte`, `DateTime`, `Decimal`, `Double`, `Int16`, `Int32`, `Int64`, `Single` or `String`
- `Decimal Floor(<type> value)` where `<type>` is one of `Decimal`, `Double`, or `Single`
- `String Lower(String value)`
- `Double Max(<type> value)` where `<type>` is one of `Byte`, `DateTime`, `Decimal`, `Double`, `Single`, `Int16`, `Int32`, `Int64`, `Single`, or `String`.

- Byte Min(<type> value) where <type> is one of Byte, DateTime, Decimal, Double, Single, Int16, Int32, Int64, Single, or String.
- Double Sum(<type> value) where <type> is one of Decimal, Double, Int16, Int32, Int64, or Single.
- String Upper(String value)

Geometry Capabilities

Use the `FdoGeometryCapabilities` object methods to learn about these capabilities. You can get this object by calling the `GetGeometryCapabilities()` method on the `FdoIConnection` object. For an explanation of the meaning of the capabilities, consult the `FdoGeometryCapabilities` class description in the FDO API Reference documentation.

Dimensionality XY is supported. The geometry component type `LinearRing` is supported. The following geometry types are supported.

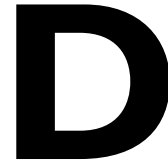
- Point
- LineString
- Polygon
- MultiPoint
- MultiLineString
- MultiPolygon

Raster Capabilities

Use the `FdoIRasterCapabilities` object methods to learn about these capabilities. You can get this object by calling the `GetRasterCapabilities()` method on the `FdoIConnection` object. For an explanation of the meaning of the capabilities, consult the `FdoIRasterCapabilities` class description in the FDO API Reference documentation.

No Raster capabilities are supported.

OSGeo FDO Provider for ODBC



This appendix discusses FDO API development issues that are related to OSGeo FDO Provider for ODBC.

What Is FDO Provider for ODBC?

The Feature Data Objects (FDO) API provides access to data in a data store. A provider is a specific implementation of the FDO API that provides access to data in a particular data store. The FDO Provider for ODBC provides FDO with access to an ODBC-based data store.

The FDO Provider for ODBC can access simple x, y, z feature objects that can run in a multi-platform environment, including Windows, Linux, and UNIX.

The FDO Provider for ODBC has the following characteristics:

- The FDO Provider for ODBC supports the definition of one or more feature classes in terms of any relational database table that contains an X, Y, and optionally, Z columns.
- Metadata, which maps the table name, and X, Y, and optionally, Z columns to a feature class, is maintained outside the database in a configuration file. This information, in conjunction with the table structure in the database, provides the definition of the feature class.
- The x, y, and z locations of objects are stored in separate properties in the primary object definition of a feature, but are accessible through a single class property 'Geometry'.

- Read-only access is provided to pre-existing data defined and populated through 3rd party applications (that is, FDO Provider for ODBC will not be responsible for defining the physical schema of the data store nor for populating the object data).
- The schema configuration of the data store is provided to the FDO Provider for ODBC through an optional XML file containing the Geographic Markup Language (GML) definition of the schema that maps ‘tables’ and ‘columns’ in the data store to feature classes and property mappings in the FDO data model.

NOTE Microsoft Excel (must have at least one named range; do not use DATABASE or other reserved words as a range name).

For more information, see *The Essential FDO* (FET_TheEssentialFDO.pdf) and the *OSGeo FDO Provider for ODBC API Reference Help* (ODBC_Provider_API.chm).

FDO Provider for ODBC Capabilities

The capabilities of an FDO provider are grouped in the following categories:

- Connection
- Schema
- Commands
- Expressions
- Filters
- Geometry
- Raster

Connection Capabilities

Use the `FdoIConnectionCapabilities` object methods to learn about these capabilities. You can get this object by calling the `GetConnectionCapabilities()` method on the `FdoIConnection` object. For an explanation of the meaning of the capabilities, consult the `FdoIConnectionCapabilities` class description in the FDO API Reference documentation.

The following capabilities are supported:

- Per connection threading
- static spatial content extent type
- SQL
- XML configuration
- Write
- Multi-user write

Schema Capabilities

Use the `FdoISchemaCapabilities` object methods to learn about these capabilities. You can get this object by calling the `GetSchemaCapabilities()` method on the `FdoIConnection` object. For an explanation of the meaning of the capabilities, consult the `FdoISchemaCapabilities` class description in the FDO API Reference documentation.

The following capabilities are supported:

- class and feature class class types
- Boolean data type with a maximum length of 1 byte
- Byte data type with a maximum length of 1 byte
- DateTime data type with a maximum length of 12 bytes
- Decimal data type with a maximum length of 56 digits (maximum decimal precision of 28 and maximum decimal scale of 28)
- Double data type with a maximum length of 8 bytes
- Int16 data type with a maximum length of 2 bytes
- Int32 data type with a maximum length of 4 bytes
- Int64 data type with a maximum length of 8 bytes
- Single data type with a maximum length of 4 bytes
- String data type with a maximum length of unknown
- Auto-generated data types Int16, Int32, and Int64
- Identity properties of type Int16, Int32, and Int64

- Name size limitation of unknown for a schema element name of type `FdoSchemaElementNameType_Datastore`
- Name size limitation of unknown for a schema element name of type `FdoSchemaElementNameType_Schema`
- Name size limitation of unknown for a schema element name of type `FdoSchemaElementNameType_Class`
- Name size limitation of unknown for a schema element name of type `FdoSchemaElementNameType_Property`
- Name size limitation of unknown for a schema element name of type `FdoSchemaElementNameType_Description`
- Characters that cannot be used for a schema element name: `::`
- Auto ID generation
- Composite ID
- Default value
- Inheritance
- Multiple schemas
- Null value constraints
- Schema overrides

Command Capabilities

Use the `FdoICommandCapabilities` object methods to learn about these capabilities. You can get this object by calling the `GetCommandCapabilities()` method on the `FdoIConnection` object. For an explanation of the meaning of the capabilities, consult the `FdoICommandCapabilities` class description in the FDO API Reference documentation.

The following commands are supported:

- `FdoCommandType_Select`
- `FdoCommandType_SelectAggregates`
- `FdoCommandType_DescribeSchema`
- `FdoCommandType_DescribeSchemaMapping`

- FdoCommandType_Insert
- FdoCommandType_Delete
- FdoCommandType_Update
- FdoCommandType_GetSpatialContexts

The following capabilities are supported:

- simple functions in Select and SelectAggregate commands
- use of expressions for properties in Select and SelectAggregates commands
- use of Distinct in SelectAggregates command
- availability of ordering in Select and SelectAggregates command
- availability of grouping criteria in SelectAggregates command

Filter Capabilities

Use the FdoIFilterCapabilities object methods to learn about these capabilities. You can get this object by calling the GetFilterCapabilities() method on the FdoIConnection object. For an explanation of the meaning of the capabilities, consult the FdoIFilterCapabilities class description in the FDO API Reference documentation.

The following capabilities are supported:

- Conditions of type comparison, like, in, null, and spatial.
- spatial operations of type Intersects, Within, Inside, and EnvelopeIntersects.

Expression Capabilities

Use the FdoIExpressionCapabilities object methods to learn about these capabilities. You can get this object by calling the GetExpressionCapabilities() method on the FdoIConnection object. For an explanation of the meaning of the capabilities, consult the FdoIExpressionCapabilities class description in the FDO API Reference documentation.

Basic and function expressions are supported.

The following functions are supported:

- Double Avg(<type> value) where <type> is one of Decimal, Double, Int16, Int32, Int53, or Single
- Decimal Ceil(<type> value) where <type> is one of Decimal, Double or Single.
- String Concat(String str1Val, String str2Val)
- Int64 Count(<type> value) where <type> is one of Boolean, Byte, DateTime, Double, Decimal, Int16, Int32, Int64, Single or String
- Decimal Floor(<type> value) where <type> is one of Decimal, Double, or Single
- String Lower(String value)
- Double Max(<type> value) where <type> is one of Byte, DateTime, Decimal, Double, Single, Int16, Int32, Int64, Single, or String.
- Byte Min(<type> value) where <type> is one of Byte, DateTime, Decimal, Double, Single, Int16, Int32, Int64, Single, or String.
- Double Sum(<type> value) where <type> is one of Decimal, Double, Int16, Int32, Int64, or Single.
- String Upper(String value)

Geometry Capabilities

Use the `FdoGeometryCapabilities` object methods to learn about these capabilities. You can get this object by calling the `GetGeometryCapabilities()` method on the `FdoIConnection` object. For an explanation of the meaning of the capabilities, consult the `FdoGeometryCapabilities` class description in the FDO API Reference documentation.

Dimensionality XYZ is supported. The Point geometry types is supported.

Raster Capabilities

Use the `FdoIRasterCapabilities` object methods to learn about these capabilities. You can get this object by calling the `GetRasterCapabilities()` method on the `FdoIConnection` object. For an explanation of the meaning of the capabilities, consult the `FdoIRasterCapabilities` class description in the FDO API Reference documentation.

No Raster capabilities are supported.

Autodesk FDO Provider for Raster



This appendix discusses FDO API development issues that are related to Autodesk FDO Provider for Raster.

What Is FDO Provider for Raster?

The Feature Data Objects (FDO) API provides access to data in a data store. A provider is a specific implementation of the FDO API that provides access to data in a particular data store. The Autodesk FDO Provider for Raster is a stand-alone file format that supports GIS data.

The FDO Provider for Raster has the following characteristics:

- The FDO Provider for Raster supports georeferenced file-based raster images and file-based grid coverages. Raster images are pixel-based images, such as digital photographs (satellite images, for example). Raster images are very useful as background images underneath your vector data, for example, an aerial photograph of a city with a layer of streets overlaying it.
- The FDO Provider for Raster can run in a multi-platform environment, including Windows and Linux.

Supported Formats

The following list shows the raster image file formats that are supported, along with their acronyms and file extensions:

- JPEG (.jpg, .jpeg) - Joint Photographic Experts Group

- JPEG2K (.jp2, .j2k) - Joint Photographic Experts Group
- MrSID (.sid) - Multi-Resolution Seamless Image Database
- PNG (.png) - Portable Network Graphic
- TIFF (.tif, .tiff) - Tagged Image File Format
- DEM (.dem) - USGS Format Digital Elevation Model
- ECW (.ecw) - Enhanced Compressed Wavelet
- DTED (.dt0, .dt1, .dt2) - Digital Terrain Elevation Data
- ESRI ASCII GRID (.asc) - ESRI Surface
- ESRI Binary GRID (.adf) - ESRI Surface

Supported Data Models

The following are the data models supported:

ModelType	BitsPerPixel	Organization	DataType
Bitonal	1	Pixel	Unsigned Integer
Grey	8	Pixel	Unsigned Integer
RGB	24	Pixel	Unsigned Integer
RGBA	32	Pixel	Unsigned Integer
Pallete	8	Pixel	Unsigned Integer
Data	1	Pixel	Unsigned Integer
Data	8	Pixel	Unsigned Integer
Data	8	Pixel	Signed Integer
Data	16	Pixel	Unsigned Integer

ModelType	BitsPerPixel	Organization	DataType
Data	16	Pixel	Signed Integer
Data	32	Pixel	Unsigned Integer
Data	32	Pixel	Signed Integer
Data	32	Pixel	Float

NOTE Only DEM, TIFF, and ECW images support the 'Data' ModelType.

NOTE All 2- and 4-BitsPerPixel images are promoted to 8 BitsPerPixel as per the underlying ATIL behavior.

For more information, see *The Essential FDO* (FET_TheEssentialFDO.pdf) and the *Autodesk FDO Provider for Raster API Reference Help* (Raster_Provider_API.chm).

FDO Provider for Raster Capabilities

The capabilities of an FDO provider are grouped in the following categories:

- Connection
- Schema
- Commands
- Expressions
- Filters
- Geometry
- Raster

Connection Capabilities

Use the `FdoIConnectionCapabilities` object methods to learn about these capabilities. You can get this object by calling the `GetConnectionCapabilities()` method on the `FdoIConnection` object. For an explanation of the meaning

of the capabilities, consult the `FdoIConnectionCapabilities` class description in the FDO API Reference documentation.

The following capabilities are supported:

- Single threaded
- static spatial content extent type
- XML configuration
- multiple spatial contexts

Schema Capabilities

Use the `FdoISchemaCapabilities` object methods to learn about these capabilities. You can get this object by calling the `GetSchemaCapabilities()` method on the `FdoIConnection` object. For an explanation of the meaning of the capabilities, consult the `FdoISchemaCapabilities` class description in the FDO API Reference documentation.

The following capabilities are supported:

- feature class class type
- String data type with a maximum length of unknown bytes
- BLOB data type with a maximum length of unknown bytes
- Identity properties of type String
- Name size limitation of unknown for a schema element name of type `FdoSchemaElementNameType_Datastore`
- Name size limitation of unknown for a schema element name of type `FdoSchemaElementNameType_Schema`
- Name size limitation of unknown for a schema element name of type `FdoSchemaElementNameType_Class`
- Name size limitation of unknown for a schema element name of type `FdoSchemaElementNameType_Property`
- Name size limitation of unknown for a schema element name of type `FdoSchemaElementNameType_Description`
- Characters that cannot be used for a schema element name: (null)
- Inheritance

- Multiple schemas
- Schema overrides

Command Capabilities

Use the `FdoICommandCapabilities` object methods to learn about these capabilities. You can get this object by calling the `GetCommandCapabilities()` method on the `FdoIConnection` object. For an explanation of the meaning of the capabilities, consult the `FdoICommandCapabilities` class description in the FDO API Reference documentation.

The following commands are supported:

- `FdoCommandType_Select`
- `FdoCommandType_SelectAggregates`
- `FdoCommandType_DescribeSchema`
- `FdoCommandType_DescribeSchemaMapping`
- `FdoCommandType_GetSpatialContexts`
- `FdoCommandType_GetMeasureUnits`

Filter Capabilities

Use the `FdoIFilterCapabilities` object methods to learn about these capabilities. You can get this object by calling the `GetFilterCapabilities()` method on the `FdoIConnection` object. For an explanation of the meaning of the capabilities, consult the `FdoIFilterCapabilities` class description in the FDO API Reference documentation.

The following capabilities are supported:

- Conditions of type in and spatial
- the Within distance operation
- spatial operations of type Within, Inside, Intersects, EnvelopeIntersects

Expression Capabilities

Use the `FdoIExpressionCapabilities` object methods to learn about these capabilities. You can get this object by calling the `GetExpressionCapabilities()` method on the `FdoIConnection` object. For an explanation of the meaning

of the capabilities, consult the `FdoIExpressionCapabilities` class description in the FDO API Reference documentation.

Function expressions are supported.

The following functions are supported:

- `BLOB MOSAIC(BLOB raster)`
- `BLOB CLIP(BLOB raster, Double minX, Double minY, Double maxX, Double maxY)`
- `BLOB RESAMPLE(BLOB raster, Double minX, Double minY, Double maxX, Double maxY, Int32 height, Int32 width)`
- `GeometricProperty SpatialExtents(BLOB raster)`

Geometry Capabilities

Use the `FdoIGeometryCapabilities` object methods to learn about these capabilities. You can get this object by calling the `GetGeometryCapabilities()` method on the `FdoIConnection` object. For an explanation of the meaning of the capabilities, consult the `FdoIGeometryCapabilities` class description in the FDO API Reference documentation.

Dimensionality XY is supported.

Raster Capabilities

Use the `FdoIRasterCapabilities` object methods to learn about these capabilities. You can get this object by calling the `GetRasterCapabilities()` method on the `FdoIConnection` object. For an explanation of the meaning of the capabilities, consult the `FdoIRasterCapabilities` class description in the FDO API Reference documentation.

Supports raster, stitching, and subsampling. Supports the following raster data models:

- Bitonal/1-bit/pixel/Unsigned Integer
- Gray/8-bit/pixel/Unsigned Integer
- RGB/24-bit/pixel/Unsigned Integer
- RGBA/32-bit/pixel/Unsigned Integer
- Palette/8-bit/pixel/Unsigned Integer

- Data/16-bit/pixel/Unsigned Integer
- Data/16-bit/pixel/Signed Integer
- Data/32-bit/pixel/Unsigned Integer
- Data/32-bit/pixel/Signed Integer
- Data/32-bit/pixel/Float

OSGeo FDO Provider for SDF



This appendix discusses FDO API development issues that are related to OSGeo FDO Provider for SDF.

What Is FDO Provider for SDF?

The Feature Data Objects (FDO) API provides access to data in a data store. A provider is a specific implementation of the FDO API that provides access to data in a particular data store. The FDO Provider for SDF is a standalone file format that supports GIS data.

The FDO Provider for SDF uses Autodesk's spatial database format, which is a file-based personal geodatabase that supports multiple features/attributes, spatial indexing, interoperability, file-locking, and high performance for large data sets.

The SDF file format has the following characteristics:

- SDF files can be read on different platforms.
- The SDF file has its own spatial indexing.
- SDF files can store geometric and non-geometric data with minimum overhead.
- Although it does not support concurrency control (locking), the SDF file format is a valid alternative to RDBMS.

For more information, see *The Essential FDO* (FET_TheEssentialFDO.pdf) and the *OSGeo FDO Provider for SDF API Reference Help* (SDF_Provider_API.chm).

FDO Provider for SDF Capabilities

The capabilities of an FDO provider are grouped in the following categories:

- Connection
- Schema
- Commands
- Expressions
- Filters
- Geometry
- Raster

Connection Capabilities

Use the `FdoIConnectionCapabilities` object methods to learn about these capabilities. You can get this object by calling the `GetConnectionCapabilities()` method on the `FdoIConnection` object. For an explanation of the meaning of the capabilities, consult the `FdoIConnectionCapabilities` class description in the FDO API Reference documentation.

The following capabilities are supported:

- Per command threaded
- dynamic spatial content extent type
- Write
- Flush

Schema Capabilities

Use the `FdoISchemaCapabilities` object methods to learn about these capabilities. You can get this object by calling the `GetSchemaCapabilities()` method on the `FdoIConnection` object. For an explanation of the meaning of the capabilities, consult the `FdoISchemaCapabilities` class description in the FDO API Reference documentation.

The following capabilities are supported:

- class and feature class class types

- Boolean data type with a maximum length of 1 byte
- Byte data type with a maximum length of 1 byte
- DateTime data type with a maximum length of 12 bytes
- Decimal data type with a maximum length of unknown digits (maximum decimal precision of unknown and maximum decimal scale of unknown)
- Double data type with a maximum length of 8 bytes
- Int16 data type with a maximum length of 2 bytes
- Int32 data type with a maximum length of 4 bytes
- Int64 data type with a maximum length of 8 bytes
- Single data type with a maximum length of 4 bytes
- String data type with a maximum length of unknown
- Int32 auto-generated data type
- Identity properties of type Boolean, Byte, DateTime, Decimal, Double, Int16, Int32, Int64, Single, and String
- Name size limitation of 255 for a schema element name of type `FdoSchemaElementNameType_Datastore`
- Name size limitation of unknown for a schema element name of type `FdoSchemaElementNameType_Schema`
- Name size limitation of unknown for a schema element name of type `FdoSchemaElementNameType_Class`
- Name size limitation of unknown for a schema element name of type `FdoSchemaElementNameType_Property`
- Name size limitation of unknown for a schema element name of type `FdoSchemaElementNameType_Description`
- Characters that cannot be used for a schema element name: `::`
- Association properties
- Auto ID generation
- Composite ID
- Exclusive value range constraints

- Inclusive value range constraints
- Inheritance
- Null value constraints
- Schema modification
- Value constraints list

Command Capabilities

Use the `FdoICommandCapabilities` object methods to learn about these capabilities. You can get this object by calling the `GetCommandCapabilities()` method on the `FdoIConnection` object. For an explanation of the meaning of the capabilities, consult the `FdoICommandCapabilities` class description in the FDO API Reference documentation.

The following commands are supported:

- `FdoCommandType_Select`
- `FdoCommandType_SelectAggregates`
- `FdoCommandType_Insert`
- `FdoCommandType_Delete`
- `FdoCommandType_Update`
- `FdoCommandType_DescribeSchema`
- `FdoCommandType_ApplySchema`
- `FdoCommandType_CreateSpatialContext`
- `FdoCommandType_GetSpatialContexts`
- `FdoCommandType_CreateDataStore`
- `FdoCommandType_DestroyDataStore`
- `SdfCommandType_ExtendedSelect`
- `SdfCommandType_CreateSDFFile`

The following capabilities are supported:

- simple functions in `Select` and `SelectAggregate` commands

- use of expressions for properties in Select and SelectAggregates commands
- use of Distinct in SelectAggregates command

Filter Capabilities

Use the `FdoIFilterCapabilities` object methods to learn about these capabilities. You can get this object by calling the `GetFilterCapabilities()` method on the `FdoIConnection` object. For an explanation of the meaning of the capabilities, consult the `FdoIFilterCapabilities` class description in the FDO API Reference documentation.

The following capabilities are supported:

- Conditions of type comparison, like, in, null, and spatial.
- spatial operations of type Contains, Disjoint, Inside, Intersects, Within, and EnvelopeIntersects

Expression Capabilities

Use the `FdoIExpressionCapabilities` object methods to learn about these capabilities. You can get this object by calling the `GetExpressionCapabilities()` method on the `FdoIConnection` object. For an explanation of the meaning of the capabilities, consult the `FdoIExpressionCapabilities` class description in the FDO API Reference documentation.

Basic and function expressions are supported.

The following functions are supported:

- `GeometricProperty SpatialExtents(GeometricProperty geomValue)`
- `Double Avg(<type> value)` where <type> is one of Byte, Decimal, Double, Int16, Int32, Int64, or Single.
- `Double Ceil(<type> value)` where <type> is one of Decimal, Double, or Single
- `String Concat(String str1Val, String str2Val)`
- `Int64 Count(<type> value)` where <type> is one of Boolean, Byte, DateTime, Decimal, Double, Int16, Int32, Int64, Single, String, BLOB, CLOB, GeometricProperty, AssociationProperty
- `Double Floor(<type> value)` where <type> is one of Decimal, Double, or Single

- String Lower(String value)
- Double Max(<type> value) where <type> is one of Byte, Decimal, Double, Int16, Int32, Int64, or Single.
- Byte Min(<type> value) where <type> is one of Byte, Decimal, Double, Int16, Int32, Int64, or Single.
- Double Sum(<type> value) where <type> is one of Byte, Decimal, Double, Int16, Int32, Int64, or Single.
- String Upper(String value)

Geometry Capabilities

Use the `FdoGeometryCapabilities` object methods to learn about these capabilities. You can get this object by calling the `GetGeometryCapabilities()` method on the `FdoIConnection` object. For an explanation of the meaning of the capabilities, consult the `FdoGeometryCapabilities` class description in the FDO API Reference documentation.

Dimensionality XYZM is supported. The geometry component types `Ring`, `LinearRing`, `CircularArcSegment`, and `LineStringSegment` are supported. The following geometry types are supported.

- Point
- LineString
- Polygon
- MultiPoint
- MultiLineString
- MultiPolygon
- MultiGeometry
- CurveString
- CurvePolygon
- MultiCurveString
- MultiCurvePolygon

Raster Capabilities

Use the `FdoIRasterCapabilities` object methods to learn about these capabilities. You can get this object by calling the `GetRasterCapabilities()` method on the `FdoIConnection` object. For an explanation of the meaning of the capabilities, consult the `FdoIRasterCapabilities` class description in the FDO API Reference documentation.

No Raster capabilities are supported.

OSGeo FDO Provider for SHP



This appendix discusses FDO API development issues that are related to OSGeo FDO Provider for SHP.

What Is FDO Provider for SHP?

The Feature Data Objects (FDO) API provides access to data in a data store. A provider is a specific implementation of the FDO API that provides access to data in a particular data store. The FDO Provider for SHP provides FDO with access to an SHP-based data store.

The FDO Provider for SHP uses a standalone file format that supports GIS data. The FDO Provider for SHP (Shape) has the following characteristics:

- Read-only access is provided to pre-existing spatial and attribute data from an Environmental Systems Research Institute (ESRI) Shape file (SHP).
- The FDO Provider for SHP can run in a multi-platform environment, including Windows and Linux.
- A Shape file consists of three separate files: SHP (shape geometry), SHX (shape index), and DBF (shape attributes in dBASE format).
- The FDO Provider for SHP accesses the information in each of the three separate files, and treats each SHP, and its associated DBF file, as a feature class with a single geometry property, and optionally, with data attribute properties.
- Schema configuration of the data store is provided to the FDO Provider for SHP through an XML file containing the Geographic Markup Language

(GML) definition of the schema that maps SHP and DBF data in the data store to feature classes and property mappings in the FDO data model.

- Although it does not support concurrency control (locking), the SHP file format is a valid alternative to RDBMS.

For more information, see *The Essential FDO* (FET_TheEssentialFDO.pdf) and the *OSGeo FDO Provider for SHP API Reference Help* (SHP_Provider_API.chm).

FDO Provider for SHP Capabilities

The capabilities of an FDO provider are grouped in the following categories:

- Connection
- Schema
- Commands
- Expressions
- Filters
- Geometry
- Raster

Connection Capabilities

Use the `FdoIConnectionCapabilities` object methods to learn about these capabilities. You can get this object by calling the `GetConnectionCapabilities()` method on the `FdoIConnection` object. For an explanation of the meaning of the capabilities, consult the `FdoIConnectionCapabilities` class description in the FDO API Reference documentation.

The following capabilities are supported:

- Per connection threading
- static spatial content extent type
- XML configuration
- multiple spatial contexts
- Write

- Multi-user write
- Flush

Schema Capabilities

Use the `FdoISchemaCapabilities` object methods to learn about these capabilities. You can get this object by calling the `GetSchemaCapabilities()` method on the `FdoIConnection` object. For an explanation of the meaning of the capabilities, consult the `FdoISchemaCapabilities` class description in the FDO API Reference documentation.

The following capabilities are supported:

- class and feature class class types
- Boolean data type with a maximum length of 1 byte
- DateTime data type with a maximum length of 12 bytes
- Decimal data type with a maximum length of 255 digits (maximum decimal precision of 255 and maximum decimal scale of 255)
- Int32 data type with a maximum length of 4 bytes
- String data type with a maximum length of 255
- Int32 auto-generated data type
- Identity properties of type Int32
- Name size limitation of 255 for a schema element name of type `FdoSchemaElementNameType_Datastore`
- Name size limitation of 7 for a schema element name of type `FdoSchemaElementNameType_Schema`
- Name size limitation of 255 for a schema element name of type `FdoSchemaElementNameType_Class`
- Name size limitation of 11 for a schema element name of type `FdoSchemaElementNameType_Property`
- Name size limitation of 0 for a schema element name of type `FdoSchemaElementNameType_Description`
- Characters that cannot be used for a schema element name: `::`
- Auto ID generation

- Multiple schemas
- Null value constraints
- Schema modification
- Schema overrides

Command Capabilities

Use the `FdoICommandCapabilities` object methods to learn about these capabilities. You can get this object by calling the `GetCommandCapabilities()` method on the `FdoIConnection` object. For an explanation of the meaning of the capabilities, consult the `FdoICommandCapabilities` class description in the FDO API Reference documentation.

The following commands are supported:

- `FdoCommandType_Select`
- `FdoCommandType_SelectAggregates`
- `FdoCommandType_Insert`
- `FdoCommandType_Delete`
- `FdoCommandType_Update`
- `FdoCommandType_DescribeSchema`
- `FdoCommandType_DescribeSchemaMapping`
- `FdoCommandType_ApplySchema`
- `FdoCommandType_DestroySchema`
- `FdoCommandType_CreateSpatialContext`
- `FdoCommandType_GetSpatialContexts`
- `SdfCommandType_CreateSDFFile`

The following capabilities are supported:

- simple functions in `Select` and `SelectAggregate` commands
- use of expressions for properties in `Select` and `SelectAggregates` commands
- use of `Distinct` in `SelectAggregates` command

Filter Capabilities

Use the `FdoIFilterCapabilities` object methods to learn about these capabilities. You can get this object by calling the `GetFilterCapabilities()` method on the `FdoIConnection` object. For an explanation of the meaning of the capabilities, consult the `FdoIFilterCapabilities` class description in the FDO API Reference documentation.

The following capabilities are supported:

- Conditions of type comparison, like, in, null, and spatial
- spatial operations of type Within, Inside, Intersects, EnvelopeIntersects

Expression Capabilities

Use the `FdoIExpressionCapabilities` object methods to learn about these capabilities. You can get this object by calling the `GetExpressionCapabilities()` method on the `FdoIConnection` object. For an explanation of the meaning of the capabilities, consult the `FdoIExpressionCapabilities` class description in the FDO API Reference documentation.

Basic and function expressions are supported.

The following functions are supported:

- `Double Avg(<type> value)` where `<type>` is one of Byte, Decimal, Double, Single, Int16, Int32, or Int64.
- `Double Ceil(<type> value)` where `<type>` is one of Decimal, Double, or Single
- `String Concat(String str1Val, String str2Val)`
- `Int64 Count(<type> value)` where `<type>` is one of Boolean, Byte, DateTime, Decimal, Double, Int16, Int32, Int64, Single, String, BLOB, CLOB, ObjectProperty, GeometricProperty, AssociationProperty, or RasterProperty
- `Decimal Floor(<type> value)` where `<type>` is one of Decimal, Double, or Single
- `String Lower(String value)`
- `Double Max(<type> value)` where `<type>` is one of Byte, Decimal, Double, Int16, Int32, Int64, or Single
- `Byte Min(<type> value)` where `<type>` is one of Byte, Decimal, Double, Int16, Int32, Int64, or Single

- Double Sum(<type> value) where <type> is one of Byte, Decimal, Double, Int16, Int32, Int64, or Single.
- String Upper(String value)
- GeometricProperty SpatialContexts(GeometricProperty property)

Geometry Capabilities

Use the `FdoGeometryCapabilities` object methods to learn about these capabilities. You can get this object by calling the `GetGeometryCapabilities()` method on the `FdoIConnection` object. For an explanation of the meaning of the capabilities, consult the `FdoGeometryCapabilities` class description in the FDO API Reference documentation.

Dimensionality XYZM is supported. The geometry component types `LinearRing` and `LineStringSegment` are supported. The following geometry types are supported.

- Point
- LineString
- Polygon
- MultiPoint
- MultiLineString

Raster Capabilities

Use the `FdoIRasterCapabilities` object methods to learn about these capabilities. You can get this object by calling the `GetRasterCapabilities()` method on the `FdoIConnection` object. For an explanation of the meaning of the capabilities, consult the `FdoIRasterCapabilities` class description in the FDO API Reference documentation.

No Raster capabilities are supported.

Autodesk FDO Provider for SQL Server



This appendix discusses FDO API development issues that are related to Autodesk FDO Provider for SQL Server.

What Is FDO Provider for SQL Server?

The Feature Data Objects (FDO) API provides access to data in a data store. A provider is a specific implementation of the FDO API that provides access to data in a particular data store. The FDO Provider for SQL Server provides FDO with access to a Microsoft SQL Server-based data store.

The Autodesk FDO Provider for SQL Server API provides custom commands that are specifically designed to work with the FDO API. For example, using these commands, you can do the following:

- Read and create schema.
- Read and write geospatial and non-geospatial data.

For more information, see *The Essential FDO* (FET_TheEssentialFDO.pdf) and the *Autodesk FDO Provider for SQL Server API Reference Help* (SQLServer_Provider_API.chm).

FDO Provider for SQL Server Capabilities

The capabilities of an FDO provider are grouped in the following categories:

- Connection

- Schema
- Commands
- Expressions
- Filters
- Geometry
- Raster

Connection Capabilities

Use the `FdoIConnectionCapabilities` object methods to learn about these capabilities. You can get this object by calling the `GetConnectionCapabilities()` method on the `FdoIConnection` object. For an explanation of the meaning of the capabilities, consult the `FdoIConnectionCapabilities` class description in the FDO API Reference documentation.

The following capabilities are supported:

- Per connection threaded
- static spatial content extent type
- transactions
- SQL
- multiple spatial contexts
- Write
- Multi-user write

Schema Capabilities

Use the `FdoISchemaCapabilities` object methods to learn about these capabilities. You can get this object by calling the `GetSchemaCapabilities()` method on the `FdoIConnection` object. For an explanation of the meaning of the capabilities, consult the `FdoISchemaCapabilities` class description in the FDO API Reference documentation.

The following capabilities are supported:

- class and feature class class types

- Boolean data type with a maximum length of 1 byte
- Byte data type with a maximum length of 1 byte
- DateTime data type with a maximum length of 12 bytes
- Decimal data type with a maximum length of 76 digits (maximum decimal precision of 38 and maximum decimal scale of 38)
- Double data type with a maximum length of 8 bytes
- Int16 data type with a maximum length of 2 bytes
- Int32 data type with a maximum length of 4 bytes
- Int64 data type with a maximum length of 8 bytes
- Single data type with a maximum length of 4 bytes
- String data type with a maximum length of 2147483647
- Int64 auto-generated data type
- Identity properties of type Boolean, Byte, DateTime, Decimal, Double, Int16, Int32, Int64, Single, and String
- Name size limitation of 123 for a schema element name of type FdoSchemaElementType_Datastore
- Name size limitation of 200 for a schema element name of type FdoSchemaElementType_Schema
- Name size limitation of 200 for a schema element name of type FdoSchemaElementType_Class
- Name size limitation of 255 for a schema element name of type FdoSchemaElementType_Property
- Name size limitation of 255 for a schema element name of type FdoSchemaElementType_Description
- Characters that cannot be used for a schema element name: .:
- Association properties
- Auto ID generation
- Composite ID
- Composite unique value constraints

- Datastore scope unique ID generation
- Default value
- Exclusive value range constraints
- Inclusive value range constraints
- Inheritance
- Multiple schemas
- Null value constraints
- Object properties
- Unique value constraints
- Schema modification
- Schema overrides
- Unique value constraints
- Value constraints list

Command Capabilities

Use the `FdoICommandCapabilities` object methods to learn about these capabilities. You can get this object by calling the `GetCommandCapabilities()` method on the `FdoIConnection` object. For an explanation of the meaning of the capabilities, consult the `FdoICommandCapabilities` class description in the FDO API Reference documentation.

The following commands are supported:

- `FdoCommandType_Select`
- `FdoCommandType_SelectAggregates`
- `FdoCommandType_Insert`
- `FdoCommandType_Delete`
- `FdoCommandType_Update`
- `FdoCommandType_DescribeSchema`
- `FdoCommandType_DescribeSchemaMapping`

- FdoCommandType_ApplySchema
- FdoCommandType_DestroySchema
- FdoCommandType_ActivateSpatialContext
- FdoCommandType_CreateSpatialContext
- FdoCommandType_DestroySpatialContext
- FdoCommandType_GetSpatialContexts
- FdoCommandType_CreateDataStore
- FdoCommandType_DestroyDataStore
- FdoCommandType_ListDataStores
- FdoCommandType_SQLCommand
- FdoRdbmsCommandType_CreateSpatialIndex
- FdoRdbmsCommandType_DestroySpatialIndex
- FdoRdbmsCommandType_GetSpatialIndexes

The following capabilities are supported:

- simple functions in Select and SelectAggregate commands
- use of expressions for properties in Select and SelectAggregates commands
- use of Distinct in SelectAggregates command
- availability of ordering in Select and SelectAggregates command
- availability of grouping criteria in SelectAggregates command

Filter Capabilities

Use the `FdoIFilterCapabilities` object methods to learn about these capabilities. You can get this object by calling the `GetFilterCapabilities()` method on the `FdoIConnection` object. For an explanation of the meaning of the capabilities, consult the `FdoIFilterCapabilities` class description in the FDO API Reference documentation.

The following capabilities are supported:

- Conditions of type comparison, like, in, null, spatial, and distance

- spatial operations of type Inside, Intersects, EnvelopeIntersects

Expression Capabilities

Use the `FdoIExpressionCapabilities` object methods to learn about these capabilities. You can get this object by calling the `GetExpressionCapabilities()` method on the `FdoIConnection` object. For an explanation of the meaning of the capabilities, consult the `FdoIExpressionCapabilities` class description in the FDO API Reference documentation.

Basic, function, and parameter expressions are supported.

The following functions are supported:

- `Double Avg(<type> value)` where <type> is one of Decimal, Double, Single, Int16, Int32, or Int64.
- `Decimal Ceil(<type> value)` where <type> is one of Decimal, Double, or Single
- `String Concat(String str1Val, String str2Val)`
- `Int64 Count(<type> value)` where <type> is one of Boolean, Byte, DateTime, Decimal, Double, Int16, Int32, Int64, Single, or String
- `Decimal Floor(<type> value)` where <type> is one of Decimal, Double, or Single
- `String Lower(String value)`
- `Byte Max(<type> value)` where <type> is one of Byte, DateTime, Decimal, Double, Int16, Int32, Int64, Single, or String.
- `Byte Min(<type> value)` where <type> is one of Byte, DateTime, Decimal, Double, Int16, Int32, Int64, Single, or String.
- `Double Sum(<type> value)` where <type> is one of Decimal, Double, Int16, Int32, Int64, or Single.
- `String Upper(String value)`

Geometry Capabilities

Use the `FdoIGeometryCapabilities` object methods to learn about these capabilities. You can get this object by calling the `GetGeometryCapabilities()` method on the `FdoIConnection` object. For an explanation of the meaning

of the capabilities, consult the `FdoIGeometryCapabilities` class description in the FDO API Reference documentation.

Dimensionality XYZM is supported. The geometry component types `Ring`, `LinearRing`, `CircularArcSegment`, and `LineStringSegment` are supported. The following geometry types are supported.

- `Point`
- `LineString`
- `Polygon`
- `MultiPoint`
- `MultiLineString`
- `MultiPolygon`
- `CurveString`
- `CurvePolygon`
- `MultiCurveString`
- `MultiCurvePolygon`

Raster Capabilities

Use the `FdoIRasterCapabilities` object methods to learn about these capabilities. You can get this object by calling the `GetRasterCapabilities()` method on the `FdoIConnection` object. For an explanation of the meaning of the capabilities, consult the `FdoIRasterCapabilities` class description in the FDO API Reference documentation.

No Raster capabilities are supported.

OSGeo FDO Provider for WFS



This appendix discusses FDO API development issues that are related to OSGeo FDO Provider for WFS.

What Is FDO Provider for WFS?

The Feature Data Objects (FDO) API provides access to data in a data store. A provider is a specific implementation of the FDO API that provides access to data in a particular data store. The FDO Provider for WFS provides FDO with access to a WFS-based data store.

An OGC Web Feature Service (WFS) provides access to geographic features that are stored in an opaque data store in a client/server environment. A client uses WFS to retrieve geospatial data that is encoded in Geography Markup Language (GML) from a single or multiple Web Feature Service. The communication between client and server is encoded in XML. If the WFS response includes feature geometries, it is encoded in Geography Markup Language (GML), which is specified in the OpenGIS Geographic Markup Language Implementation Specification.

Using FDO Provider for WFS data manipulation operations, you can do the following:

- Query features based on spatial and non-spatial constraints.
- Create new feature instances.
- Delete feature instances.
- Update feature instances.

- Lock feature instances.

For more information, see *The Essential FDO* (FET_TheEssentialFDO.pdf).

NOTE There is no public API documentation for the FDO Provider for WFS; functionality is available through the main FDO API.

FDO Provider for WFS Capabilities

The capabilities of an FDO provider are grouped in the following categories:

- Connection
- Schema
- Commands
- Expressions
- Filters
- Geometry
- Raster

Connection Capabilities

Use the `FdoIConnectionCapabilities` object methods to learn about these capabilities. You can get this object by calling the `GetConnectionCapabilities()` method on the `FdoIConnection` object. For an explanation of the meaning of the capabilities, consult the `FdoIConnectionCapabilities` class description in the FDO API Reference documentation.

The following capabilities are supported:

- Per connection threaded
- static spatial content extent type
- multiple spatial contexts

Schema Capabilities

Use the `FdoISchemaCapabilities` object methods to learn about these capabilities. You can get this object by calling the `GetSchemaCapabilities()`

method on the `FdoIConnection` object. For an explanation of the meaning of the capabilities, consult the `FdoISchemaCapabilities` class description in the FDO API Reference documentation.

The following capabilities are supported:

- class and feature class class types
- Boolean data type with a maximum length of unknown bytes
- Byte data type with a maximum length of unknown bytes
- DateTime data type with a maximum length of unknown bytes
- Decimal data type with a maximum length of unknown digits (maximum decimal precision of unknown and maximum decimal scale of unknown)
- Double data type with a maximum length of unknown bytes
- Int16 data type with a maximum length of unknown bytes
- Int32 data type with a maximum length of unknown bytes
- Int64 data type with a maximum length unknown 8 bytes
- Single data type with a maximum length of unknown bytes
- String data type with a maximum length of unknown
- Identity properties of type Boolean, Byte, DateTime, Decimal, Double, Int16, Int32, Int64, Single, and String
- Name size limitation of unknown for a schema element name of type `FdoSchemaElementNameType_Datastore`
- Name size limitation of unknown for a schema element name of type `FdoSchemaElementNameType_Schema`
- Name size limitation of unknown for a schema element name of type `FdoSchemaElementNameType_Class`
- Name size limitation of unknown for a schema element name of type `FdoSchemaElementNameType_Property`
- Name size limitation of unknown for a schema element name of type `FdoSchemaElementNameType_Description`
- Characters that cannot be used for a schema element name: (null)
- Composite ID

- Multiple schemas
- Object properties

Command Capabilities

Use the `FdoICommandCapabilities` object methods to learn about these capabilities. You can get this object by calling the `GetCommandCapabilities()` method on the `FdoIConnection` object. For an explanation of the meaning of the capabilities, consult the `FdoICommandCapabilities` class description in the FDO API Reference documentation.

The following commands are supported:

- `FdoCommandType_Select`
- `FdoCommandType_SelectAggregates`
- `FdoCommandType_DescribeSchema`
- `FdoCommandType_GetSpatialContexts`
- `FdoRdbmsCommandType_GetSpatialIndexes`

The following capabilities are supported:

- use of expressions for properties in `Select` and `SelectAggregates` commands

Filter Capabilities

Use the `FdoIFilterCapabilities` object methods to learn about these capabilities. You can get this object by calling the `GetFilterCapabilities()` method on the `FdoIConnection` object. For an explanation of the meaning of the capabilities, consult the `FdoIFilterCapabilities` class description in the FDO API Reference documentation.

No capabilities are supported:

Expression Capabilities

Use the `FdoIExpressionCapabilities` object methods to learn about these capabilities. You can get this object by calling the `GetExpressionCapabilities()` method on the `FdoIConnection` object. For an explanation of the meaning of the capabilities, consult the `FdoIExpressionCapabilities` class description in the FDO API Reference documentation.

Basic expressions are supported.

The following functions are supported:

- `GeometricProperty SpatialExtents(GeometricProperty property)`

Geometry Capabilities

Use the `FdoIGeometryCapabilities` object methods to learn about these capabilities. You can get this object by calling the `GetGeometryCapabilities()` method on the `FdoIConnection` object. For an explanation of the meaning of the capabilities, consult the `FdoIGeometryCapabilities` class description in the FDO API Reference documentation.

Dimensionality XYZM is supported. The geometry component types `Ring`, `LinearRing`, `CircularArcSegment`, and `LineStringSegment` are supported. The following geometry types are supported.

- `Point`
- `LineString`
- `Polygon`
- `MultiPoint`
- `MultiLineString`
- `MultiPolygon`
- `MultiGeoemtry`
- `CurveString`
- `CurvePolygon`
- `MultiCurveString`
- `MultiCurvePolygon`

Raster Capabilities

Use the `FdoIRasterCapabilities` object methods to learn about these capabilities. You can get this object by calling the `GetRasterCapabilities()` method on the `FdoIConnection` object. For an explanation of the meaning of the capabilities, consult the `FdoIRasterCapabilities` class description in the FDO API Reference documentation.

No Raster capabilities are supported.

OSGeo FDO Provider for WMS



This appendix discusses FDO API development issues that are related to OSGeo FDO Provider for WMS.

What Is FDO Provider for WMS?

The Feature Data Objects (FDO) API provides access to data in a data store. A provider is a specific implementation of the FDO API that provides access to data in a particular data store. The FDO Provider for WMS provides FDO with access to a WMS-based data store.

An Open Geospatial Consortium (OGC) Web Map Service (WMS) produces maps of spatially referenced data dynamically from geographic information. This international standard defines a "map" to be a portrayal of geographic information as a digital image file suitable for display on a computer screen. A map is not the data itself. Maps by WMS are generally rendered in a pictorial format, such as PNG, GIF or JPEG, or occasionally as vector-based graphical elements in Scalable Vector Graphics (SVG) or Web Computer Graphics Metafile (WebCGM) formats.

The FDO Provider for WMS has the following characteristics:

- The FDO Provider for WMS serves up map information originating from an OGC Basic Web Map Service that provides pictorially formatted images, such as PNG, GIF, or JPEG.
- WMS map data is exposed through an FDO feature schema whose classes contain an FDO Raster property definition. The FDO schema exposed from the FDO Provider for WMS conforms to a pre-defined FDO schema that is

specific to WMS and that acts as the basis for all FDO interaction with WMS data, regardless of the originating source of the WMS images.

- WMS data manipulation operations are limited to querying features based on spatial and non-spatial constraints. Schema manipulation operations are not supported.

The FDO Provider for WMS can run in a multi-platform environment, including Windows and Linux.

For more information, see *The Essential FDO* (FET_TheEssentialFDO.pdf) and the *OSGeo FDO Provider for WMS API Reference Help* (WMS_Provider_API.chm).

FDO Provider for WMS Capabilities

The capabilities of an FDO provider are grouped in the following categories:

- Connection
- Schema
- Commands
- Expressions
- Filters
- Geometry
- Raster

Connection Capabilities

Use the `FdoIConnectionCapabilities` object methods to learn about these capabilities. You can get this object by calling the `GetConnectionCapabilities()` method on the `FdoIConnection` object. For an explanation of the meaning of the capabilities, consult the `FdoIConnectionCapabilities` class description in the FDO API Reference documentation.

The following capabilities are supported:

- Per connection threading
- static spatial content extent type
- XML configuration

Schema Capabilities

Use the `FdoISchemaCapabilities` object methods to learn about these capabilities. You can get this object by calling the `GetSchemaCapabilities()` method on the `FdoIConnection` object. For an explanation of the meaning of the capabilities, consult the `FdoISchemaCapabilities` class description in the FDO API Reference documentation.

The following capabilities are supported:

- class and feature class class types
- String data type with a maximum length of unknown
- BLOB data type with a maximum length of unknown bytes
- Name size limitation of unknown for a schema element name of type `FdoSchemaElementNameType_Datastore`
- Name size limitation of unknown for a schema element name of type `FdoSchemaElementNameType_Schema`
- Name size limitation of unknown for a schema element name of type `FdoSchemaElementNameType_Class`
- Name size limitation of unknown for a schema element name of type `FdoSchemaElementNameType_Property`
- Name size limitation of unknown for a schema element name of type `FdoSchemaElementNameType_Description`
- Inheritance
- Schema overrides

Command Capabilities

Use the `FdoICommandCapabilities` object methods to learn about these capabilities. You can get this object by calling the `GetCommandCapabilities()` method on the `FdoIConnection` object. For an explanation of the meaning of the capabilities, consult the `FdoICommandCapabilities` class description in the FDO API Reference documentation.

The following commands are supported:

- `FdoCommandType_Select`
- `FdoCommandType_SelectAggregates`

- FdoCommandType_DescribeSchema
- FdoCommandType_DescribeSchemaMapping
- FdoCommandType_GetSpatialContexts

The following capabilities are supported:

- simple functions in Select and SelectAggregate commands

Filter Capabilities

Use the `FdoIFilterCapabilities` object methods to learn about these capabilities. You can get this object by calling the `GetFilterCapabilities()` method on the `FdoIConnection` object. For an explanation of the meaning of the capabilities, consult the `FdoIFilterCapabilities` class description in the FDO API Reference documentation.

No filter capabilities are supported:

Expression Capabilities

Use the `FdoIExpressionCapabilities` object methods to learn about these capabilities. You can get this object by calling the `GetExpressionCapabilities()` method on the `FdoIConnection` object. For an explanation of the meaning of the capabilities, consult the `FdoIExpressionCapabilities` class description in the FDO API Reference documentation.

Function expressions are supported.

The following functions are supported:

- `BLOB RESAMPLE(BLOB raster, Double minX, Double minY, Double maxX, Double maxY, Int32 height, Int32 width)`
- `BLOB CLIP(BLOB raster, Double minX, Double minY, Double maxX, Double maxY)`
- `GeometricProperty SpatialExtents(BLOB raster)`

Geometry Capabilities

Use the `FdoIGeometryCapabilities` object methods to learn about these capabilities. You can get this object by calling the `GetGeometryCapabilities()` method on the `FdoIConnection` object. For an explanation of the meaning

of the capabilities, consult the `FdoGeometryCapabilities` class description in the FDO API Reference documentation.

Dimensionality XY is supported. The geometry component type `LinearRing` is supported. The following geometry types are supported.

- Polygon

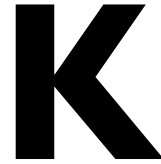
Raster Capabilities

Use the `FdoRasterCapabilities` object methods to learn about these capabilities. You can get this object by calling the `GetRasterCapabilities()` method on the `FdoIConnection` object. For an explanation of the meaning of the capabilities, consult the `FdoRasterCapabilities` class description in the FDO API Reference documentation.

The Raster capability is supported. The following raster data models are supported:

- Bitonal/1-bit/pixel/Unsigned Integer
- Gray/8-bit/pixel/Unsigned Integer
- RGB/24-bit/pixel/Unsigned Integer
- RGBA/32-bit/pixel/Unsigned Integer
- Palette/8-bit/pixel/Unsigned Integer
-

Expression Functions



Introduction

The enhanced set includes aggregate, conversion, date, mathematical, numeric, string and geometry functions. All functions are supported by all providers, with the exception of the Raster, WFS and WMS providers.

This appendix outlines the following expression functions details:

- Signatures
- Built-in support for some of the functions based on the supported RDBMS
- Provider-specific support for some of the functions
- Implementation options

Expression Function List

This section lists and describes all supported expression functions.

Aggregate Expression Functions

The following aggregate expression functions are supported:

Function Name	Description
Avg	Returns the average of the values identified by the provided expression.

Function Name	Description
Count	Returns the number of rows returned by a query identified by the provided expression.
Max	Returns the maximum value of the provided expression.
Median	Represents an inverse distribution function that assumes a continuous distribution model. It takes a numeric or date-time value and returns the middle value or an interpolated value that would be the middle value once the values are sorted.
Min	Returns the minimum value of the provided expression.
Stddev	Returns the sample standard deviation of the provided expression.
Sum	Returns the sum of the values identified by the provided expression.
Spatial Extent	Returns the spatial extent of a geometry.

Conversion Expression Functions

The following conversion expression functions are supported.

Function Name	Description
NullValue	Evaluates two given expressions and returns the first one if it does not evaluate to NULL, the second otherwise.
ToDate	Converts a string with date/time information to a date.
ToDouble	Converts a numeric or string expression to a double.
ToFloat	Converts a numeric or string expression to a float.
ToInt32	Converts a numeric or string expression to an int32.
ToInt64	Converts a numeric or string expression to an int64.

Function Name	Description
ToString	Converts a numeric or date expression to a string.

Date Expression Functions

The following date expression functions are supported:

Function Name	Description
AddMonths	Adds a specified number of months to a given date expression.
CurrentDate	Returns the current date.
Extract	Extracts a specified portion of a date.
MonthsBetween	Calculates the number of months between two provide date expressions.

Geometry Expression Functions

The following geometry expression functions are supported.

Function Name	Description
Area2D	Returns area of a geometry.
Length2D	Returns length of a geometry.

Mathematical Expression Functions

The following mathematical expression functions are supported:

Function Name	Description
Abs	Returns the absolute value of a numeric expression.
Acos	Returns the arc cosine of a numeric expression.
Asin	Returns the arc sine of a numeric expression.
Atan	Returns the arc tangent of a numeric expression.
Cos	Returns the cosine of a numeric expression.
Exp	Returns e raised to the power of a numeric expression.
Ln	Returns the natural logarithm of a numeric expression.
Log	Returns the logarithm of a numeric expression using the provided base.
Mod	Returns the remainder of the division of two numeric expressions.
Power	Returns the result of a numeric expression raised to the power of another numeric expression.
Remainder	Returns the remainder of the division of two numeric expressions.
Sin	Returns the sine of a numeric expression.
Sqrt	Returns the square root of a numeric expression.
Tan	Returns the tangent of a numeric expression.

M	N	Mod (M/N)	Classical Modulus
11	4	3	3
11	-4	3	-1
-11	4	-3	1
-11	-4	-3	-3

Figure 1: Modulus Implementation Results

Numeric Expression Functions

The following numeric expression functions are supported.

Function Name	Description
Ceil	Returns the smallest integer greater than or equal to a numeric expression.
Floor	Returns the largest integer less than or equal to a numeric expression.
Round	Returns the rounded value of a numeric expression.
Sign	Returns -1 if the provided numeric expression evaluates to a value less than 0, 0 if the expression evaluates to 0 and 1 if the expression evaluates to a value bigger than 0.
Trunc	Truncates a numeric or date expression.

String Expression Functions

The following string expression functions are supported:

The function Translate does not translate a string from one character set to a different one. Instead, it just replaces a set of letters in a string with their corresponding replacement characters where each character specified in the set of characters to be replaced is replaced by the character in the set of replacement characters at the same place. For example, if the call is Translate('SQL*Plus User's Guide', '* /', '---'), each of the specified characters in the first set will be replaced by the character in the same position in the

second set. In this example, the resulting string will be SQL_Plus_Users_Guide because:

- Any space will be replaced by a _
- Any * will be replaced by a _
- Any / will be replaced by a _ (in this case, nothing will be done because the string does not include the specified character)
- Any ' will be removed because there is no corresponding replacement character specified.

Function Name	Description
Concat	Returns the concatenation of two string expressions.
Instr	Returns the position of a substring in a string expression.
Length	Returns the length of a string expression.
Lower	Converts all uppercase letters in a string expression into lowercase letters.
Lpad	Pads a string expression to the left to a predefined string length.
Ltrim	Removes leading blanks from a string expression.
Rpad	Pads a string expression to the right to a predefined string length.
Rtrim	Removes trailing blanks from a string expression.
Soundex	Returns the phonetic representation of a string expression.
Substr	Extracts a substring from a string expression.
Translate	Replaces a set of letters in a string.
Trim	Removes leading and/or trailing blanks from a string expression.
Upper	Converts all lowercase letters in a string expression into uppercase letters.

Index

A

- AGF 112
- API 5
 - defined 5
 - FDO 5
- application development 11
- architecture and packages 8
- ArcSDE 148
 - limitations 148
- ArcSDE Provider 147, 159, 166, 174, 181, 188, 196, 201, 210, 216
 - Capabilities 159, 166, 174, 181, 188, 196, 201, 210, 216
 - Software Requirements 147
- association property 15
- Autodesk Geometry Format (AGF) 112

B

- behaviors, GisPtr 25
- BinaryOperations 106

C

- calls, chain 25
- capabilities 35–37, 40–44, 139, 159, 166, 174, 181, 188, 196, 201, 210, 216
 - API 35
 - Command 40
 - Connection 36
 - Expression 41
 - FDO Provider for ArcSDE 159, 166, 174, 181, 188, 196, 201, 210, 216
 - FDO Provider for Oracle 139
 - Filter 42
 - geometry 43
 - Raster 44
 - Schema 37

- class 14, 52–53
 - contained 52
 - feature 14
 - IdentityProperty and ObjectProperty 52–53
 - standalone 52
- Class Diagram, FDO Schema Element 54
- class type 14
- code example 36–37, 40–44
 - Command capabilities 40
 - Connection capabilities 36
 - Expression capabilities 41
 - Filter capabilities 42
 - Geometry capabilities 43
 - Raster capabilities 44
 - Schema capabilities 37
- collection classes 23
- comparison operations 106
- connection 27, 29, 124, 152
 - ArcSDE 152
 - establishing 29
 - Oracle 124
 - semantics 27
- constraints, expression text 101
- constraints, filter text 101
- constraints, provider-specific 101
- Contained Class 52
- context, spatial 18

D

- data concepts 13
- data property 15, 131–132
 - defined 15
 - overrides 131
- data sources and data stores 27
- data store 10, 18–20, 47, 89, 99, 123, 147, 165, 173, 187, 195, 215
 - defined 18
 - FDO Provider for ArcSDE 147
 - FDO Provider for MySQL 165

- FDO Provider for ODBC 173
- FDO Provider for Oracle 123
- FDO Provider for SDF 187
- FDO Provider for SHP 195
- FDO Provider for WMS 215
- filtering 19, 99
- locking 19
- Oracle 10
- querying 89
- schemas and the 47
- transactions 20
- data stores, data sources and 27
- data type 104–105
 - DATETIME 105
 - DOUBLE 105
 - IDENTIFIER 104
 - INTEGER 105
 - PARAMETER 104
 - STRING 104
- data type mappings 153
- data types 104
- DataStore 147
 - FDO Provider for ArcSDE 147
- decimal properties 133
- develop applications 11
- dimensionality, defined 15

E

- edit a GML schema file 62
- element states, schema 55
- elements of a schema 14
- example 72, 75, 82, 85, 87, 90
 - creating a schema 72
 - creating a schema read in from an XML file 75
 - deleting property values 87
 - describing a schema 75
 - destroying a schema 75
 - inserting an Integer, a string, and a Geometry Value 82
 - query 90
 - schema management 72
 - updating property values 85
- expression grammar 103
- expression text 100

- Expression, defined 19
- expressions 100

F

- factory, abstract geometry 119
- FDO 8
 - architecture and packages 8
- FDO API 5
 - defined 5
- FDO concepts 14–20
 - commands 18
 - data store 18
 - expression 19
 - feature class 14
 - filter 19
 - geometry property 16
 - locking 19
 - object property 17
 - property 15
 - spatial context 18
 - transactions 20
- FDO In General 121
- FDO Lock Types, OWM and 137
- FDO Provider for ArcSDE 147, 152, 159, 166, 174, 181, 188, 196, 201, 210, 216
 - capabilities 159, 166, 174, 181, 188, 196, 201, 210, 216
 - connection 152
 - defined 147
 - software requirements 147
- FDO Provider for MySQL 165
 - defined 165
- FDO Provider for ODBC 173
 - defined 173
- FDO Provider for Oracle 123–124, 130, 139
 - capabilities 139
 - connection 124
 - defined 123
 - schema overrides 130
 - software requirements 124
- FDO Provider for Raster 179
 - defined 179

- FDO Provider for SDF 187
 - defined 187
- FDO Provider for SDF, defined 187
- FDO Provider for SHP 195
 - defined 195
- FDO Provider for SQL Server 201
 - defined 201
- FDO Provider for WMS 215
 - defined 215
- FDO schema element class diagram 54
- FDO XML format 56
- FDOClass 51, 135
- FDOClassDefinition 134
- FDOFeatureClass 51
- FDOIActivateLongTransaction 96
- FDOICommitLongTransaction 97
- FDOICreateLongTransaction 97
- FDOIDeactivateLongTransaction 96
- FDOIGetLongTransactions 97
- FDOIRollbackLongTransaction 96
- feature class 14
- feature schema, creating a 154
- filter 101
 - grammar 101
- Filter 19, 99
 - defined 19
- filter text 100
- filters 19
- foreign schema 126
 - limitations 126
 - read-write privileges 126
- foreign schemas 125
 - FDO Provider for Oracle 125

G

- geometric property 131, 133
 - overrides 131
- geometric types, mapping between
 - Geometry and 120
- geometry 16
- Geometry 16, 107, 111–112, 119
 - basic or pure 112
 - properties 16
 - types 119
 - value 107

- working with 111
- geometry and geometric types, mapping
 - between 120
- Geometry API 111
- GIS_SAFE_RELEASE (*ptr) 21–22
- GisPtr 22
- GML schema file, creating and
 - editing 62

H

- handler, exception 23

I

- identity properties 133

K

- keywords, filter and expression 104

L

- lock types 137
- lock, long transaction exclusive 137
- locking 19, 136, 149
 - ArcSDE limitations 149
 - defined 19
- long transaction 20, 95–96, 136
 - defined 95
 - leaf 95
 - root 20, 95–96, 136
- long transactions, locking and 136

M

- mappings 48, 153
 - data type 153
 - physical 48
- memory management 21
- models, modifying 55

N

- non-feature class issues 52
- non-smart ptr 26

O

- object property 17, 131, 133
 - defined 17
 - overrides 131
- ObjectProperty types 51
- OGC WKT 111
- operations, comparison 106
- operations, data maintenance 79
- operator precedence 106
- operators 106
- Oracle 136–137
 - lock types 137
 - long transaction versions 136
- Oracle identity property 126
- Oracle provider, software requirements 124
- Oracle reserved words used with filter and expression text 135
- Oracle Workspace Manager 138
- Oracle-specific 132
 - schema creation restrictions 132
- Oracle, FDO Provider for 123
- overrides 49, 130–131
 - class table 131
 - data property 131
 - FDO Provider for Oracle and Schema 130
 - geometric property 131
 - object property 131
 - schema 49

P

- package 27, 47
 - connections 27
 - Schema 47
- packages, FDO 8
- parent in the schema classes 48
- properties 16, 47, 132–133
 - base 47

- data 132
- decimal 133
- geometric 133
- Geometry 16
- identity 133
- object 133
- string 133

- property 15, 17, 135
 - association 15
 - data 15
 - defined 15
 - object 17
 - Raster 17

- property definitions, adding GML for 66
- property overrides 131

- data 131
- geometric 131
- object 131

- property values 80–81
 - data 80
 - geometry 81

- Provider for ArcSDE 147, 152
 - connection 152
 - defined 147

- Provider for MySQL 165
 - defined 165

- Provider for ODBC 173
 - defined 173

- Provider for Oracle 123–124, 139
 - capabilities, FDO 139
 - connection, FDO 124
 - defined 123
 - general requirements 124

- Provider for Raster 179
 - defined 179

- Provider for SHP 195
 - defined 195

- Provider for SQL Server 201
 - defined 201

- provider, defined 9

Q

- query 89–90
 - creating 89
 - example 90

R

- raster property 17
 - defined 17
- references, cross-schema 48
- requirements 124, 147
 - FDO Provider for ArcSDE 147
 - FDO Provider for Oracle 124
- restrictions, Oracle-specific schema creation 132
- rollback mechanism, schema 56
- root long transaction, defined 20

S

- SampleFeatureSchema.xml 76
- schema 13–14
 - defined 13
 - schema elements 14
- schema creation restrictions,
 - Oracle-specific 132
- schema management 72
- schema mapping, defined 14
- schema modification restrictions,
 - Oracle-specific 134
- schema overrides 14, 130
- schema, create 50, 154
- schemas 49, 53, 55–56
 - describing 53
 - element states 55
 - modifying models 55
 - rollback mechanism 56
 - working with 49
- SDF 187
 - FDO Provider for 187
- software requirements 124, 147
- spatial context 120

- spatial context, defined 18
- special characters 107
- standalone class 52
- states, schema element 55
- string properties 133
- supported interfaces, LT 96

T

- table overrides, class 131
- text, expression 100
- text, filter 100
- transaction, long 20
- types 51, 119
 - Geometry 119
 - ObjectProperty 51

U

- UnaryOperations 106

V

- values 85–86
 - deleting 86
 - updating 85

W

- WKT 111
- workspace, Oracle 138

X

- XML Format, FDO 56

